Advanced Scientific Computing, B673

Zdzisław Meglicki

February 26, 2001

Contents

Note These are B673 notes from the 1999 Spring Semester. They will be revised and enhanced for the 2000 Spring Semester, e.g., some stuff that was covered in the 1999 Spring Semester is not here in these notes. Some other stuff like *Quantum Computing* is covered by a new course: B679. See *http://beige.ucs.indiana.edu/B679* for more information about that course. The time and venue for the 2000 Spring Semester course is

lecture every Monday, beginning with the 10th of January 2000, in LH019 from 11:15AM to 12:30PM

laboratory every Wednesday, beginning with the 12th of January 2000, in GY226 from 11:15AM to 12:30PM

Chapter 1

Introduction

The Advanced Scientific Computing course, B673, will cover selected topics related to scientific computing that are perhaps rightly considered *advanced*. Whereas in the previous course, Scientific Computing, P573, we have covered quite a broad range of tools and techniques, most of them quite easy to master, in this course we'll concentrate on a smaller number of somewhat more difficult topics.

The five major topics that we're going to focus on will be

1. linear algebra and fast Fourier transform packages and algorithms,
2. Message Passing Interface (MPI) and parallel I/O (MPI/IO),
3. 3D visualisation of scientific data sets,
4. implementation and architecture of problem solving environments: octave and calc,
5. quantum computing.

This is a multidisciplinary course. It is addressed to graduate students from various schools who need to develop and use advanced computational techniques in their research.

1.1 Time and Venue

Where the lectures and laboratory classes are held

We have a small lecture room, Lindley Hall 019, reserved for Mondays and Wednesdays, 11:15AM to 12:30PM. But the course will require a considerable amount of hands-on work. Consequently I have booked the Nations Laboratory in the Geology Building, Room 226 (the Ships Cluster, unfortunately, is already booked by someone else), for every Wednesday, 11:15AM to 12:30PM, in the Spring semester (cf. Table 1.1). In summary we are going to have one lecture a week, on Monday, in Lindley Hall, followed by one laboratory class a week, on Wednesday, in the Geology Building.

Monday	*lecture*	11:15AM – 12:30PM	Lindley Hall 019
Wednesday	*laboratory*	11:15AM – 12:30PM	Nations Lab, Geology Building, Room 226

Table 1.1: Time and Venue for B673

Reservation Card: 7235
B673 (Section 1421) Advanced Scientific Computing
01/10/2000 to 04/29/2000: 11:15 AM - 12:29 PM

room	*date*	*from*	*to*	*seats*	*status*
GY 226	01/12/2000	11:15 AM	12:29 PM	22	Confirmed
GY 226	01/19/2000	11:15 AM	12:29 PM	22	Confirmed
GY 226	01/26/2000	11:15 AM	12:29 PM	22	Confirmed
GY 226	02/02/2000	11:15 AM	12:29 PM	22	Confirmed
GY 226	02/09/2000	11:15 AM	12:29 PM	22	Confirmed
GY 226	02/16/2000	11:15 AM	12:29 PM	22	Confirmed
GY 226	02/23/2000	11:15 AM	12:29 PM	22	Confirmed
GY 226	03/01/2000	11:15 AM	12:29 PM	22	Confirmed
GY 226	03/08/2000	11:15 AM	12:29 PM	22	Confirmed
GY 226	03/22/2000	11:15 AM	12:29 PM	22	Confirmed
GY 226	03/29/2000	11:15 AM	12:29 PM	22	Confirmed
GY 226	04/05/2000	11:15 AM	12:29 PM	22	Confirmed
GY 226	04/12/2000	11:15 AM	12:29 PM	22	Confirmed
GY 226	04/19/2000	11:15 AM	12:29 PM	22	Confirmed
GY 226	04/26/2000	11:15 AM	12:29 PM	22	Confirmed

Table 1.2: Reservation Card 7235 for B673, Advanced Scientific Computing, Spring Semester 2000

Table 1.2 has been extracted from the STC Lab Reservation Note.

In summary we should look forward to 15 laboratory classes (we lose a class on the 15th of March due to Spring break), and 13 lectures (we lose a lecture on the 17th of Januray, it's the Martin Luther King Day, and then a lecture on the 13th of March due to Spring break). See the Bulletin Board (section 1.8) for any other information.

1.2 Expected Level of Skills

What you need to know to attend the course

You should know the basics of algebra and analysis: matrix manipulations, determinants, vector spaces, derivatives, integrals, differential equations.

You will need to be acquainted with code and data maintenance utilities such as `make`, `RCS`, `FWEB`, and `HDF`, as well as with Fortran-90 (F90) and High Performance Fortran (HPF).

You should know something about Emacs and Emacs Calc, Maxima (or

Maple) and Octave (or Matlab), and, last but not least, you should know how to work with the Indiana University SP supercomputer, i.e., how to submit parallel jobs to Load Leveler and how to run them interactively, how to link subroutines from the PESSL library and from the HDF library with your HPF programs, and how to work with AFS on the SP and on the Ships cluster in the Student Building. Basic UNIX skills are a must.

If any of these are a novelty to you, you can learn about them by going through

- "Introduction to Scientific Computing, P573, Course Notes",
- "LoadLeveler Crib"
- "Load Leveler Hints and Recipes"
- "How to Time, Save, and Resubmit LoadLeveler Jobs"
- "XL Fortran for AIX, Language Reference, Version 4 Release 1"
- "XL High Performance Fortran for AIX, Language Reference, Version 1 Release 1 "
- "Parallel Engineering and Scientific Subroutine Library, Guide and Reference, Release 2"

A significant dose of hand-holding will be provided to less experienced students, so that even if you're not very skilled in some of the above you should be able to get through the course without much difficulty.

1.3 The Syllabus

The general syllabus for this course is a continuation of what has already been covered in P573 and what is required by the PhD Qualifying Exam.

The Syllabus for the Qualifying Exam

Here is the syllabus itself:

1. Basic computer concepts and tools
   (a) Measuring time on computers (cf. "How to Time, Save, and Resubmit LoadLeveler Jobs"
      i. wall clock, user, system times
      ii. clock resolution and overhead of timing
      iii. other basic performance data: page faults, block I/O operations, memory/space integrals
   (b) Data sets and characteristics: ASCII, binary, sequential, random access, record addressability (cf. P573)
   (c) floating point data: IEEE standard, range and precision, infinity, +0 and -0, denormalised numbers, NaN (cf. P573)

2. High performance computer architecture
   (a) Organisation of computer memory hierarchy
      i. Memory banks and interleaving
      ii. Caches and registers
         A. Direct mapped, set associative
         B. Cache lines
         C. Replacement policies
         D. Write-through and write-back
         E. Snooping
      iii. Enhancing data locality in codes
      iv. I/O architectures
   (b) Microprocessors
      i. CISC versus RISC versus EPISC (Merced)
      ii. Pipelining of data and instructions
      iii. Superscalar organisation
      iv. Address computations and pointers
   (c) Supercomputer organisation
      i. Vectorisation (cf. P573)
      ii. Shared memory, distributed memory
      iii. Task versus data parallelism: SIMD versus MIMD
      iv. Topologies: mesh, hypercube, tree
      v. Synchronisation and communication: MPI, PVM, blocking communication, broadcasting
      vi. Examples: CM2, CM5, SGI PC, Sun E10000, SP2, Cray T3E, SGI O2000, HP Exemplar, Fujitsu VPP300, NEC SV6
3. Numerical methods overview (here the emphasis is on implications for data structures and mapping of computations to machine architectures and less on the mathematical analysis of the methods)
   (a) Common models, prototypes and implications for computer codes: for each of these be able to discuss implementation issues, choice of data structures, performance prediction, impact on structure of computational kernels (cf. P573)
      i. heat equation: parabolic PDEs
      ii. wave equation: hyperbolic PDEs
      iii. laplace equation: elliptic PDEs
      iv. planetary motion: systems of ODEs
      v. double pendulum: chaotic ODEs
      vi. N-body systems: particle methods, reduced order methods (e.g., Barnes-Hut, multipole)

   vii. signal processing: Fourier analysis (FFT, butterfly pattern, cf. P573)

   (b) Discretisations (cf. P573)

      i. time discretisation: explicit versus implicit methods
      ii. spatial discretisation: finite differences, finite elements, finite volumes, spectral elements
      iii. uniform and quasi-uniform meshes
      iv. irregular and adaptive meshing
      v. integral equation methods

   (c) Sparse matrix data structures and their manipulation

      i. operations needed on sparse matrices: matrix-vector products, Gaussian elimination, triangular solvers
      ii. coordinatewise, compressed sparse row, modified sparse row, jagged diagonals
      iii. load/store analysis and pipelining for sparse matrix data structures

4. Performance analysis and improvement

   (a) Profiling

      i. instrumentation and sampling-based tools: *gprof*, *tprof*, *pixie*, CASE tools
      ii. interpreting profiling information

   (b) Benchmarking, MFLOPS, MIPS, theoretical peak performance

   (c) Analysing and improving performance

      i. using compiler optimisations
      ii. typical techniques: common sense, loop interchange, unrolling, splitting, blocking, jamming
      iii. validation of results

5. Programming language and systems issues in scientific computing

   (a) Fortran 90 concepts: vector and array operators, modules and interfaces, operators and when and how to use them (cf. P573)
   (b) data parallelism in High Performance Fortran (cf. P573)
   (c) languages for interactive scientific experiments: Matlab, Mathematica, Maple (cf. P573)
   (d) object oriented scientific programming techniques
   (e) understanding libraries: templates, macros, BLAS, LAPACK, and related resources (cf. P573)
   (f) the role of the programmer in understanding the compiler, preprocessors and optimisers

(g) programming support for large scientific data bases (cf. P573)

(h) software support for parallel programs (cf. P573)

i. parallelising compiler techniques: synchronisation methods, types of data dependencies, compiler directives
ii. communicating processes: PVM, MPI
iii. POSIX threads (Pthreads)

6. Parallelism in scientific algorithms

(a) Modeling of parallelism in theory and practice

i. speed-up
ii. parallel efficiency
iii. Amdahl's law
iv. computation/communication ratio

(b) Parallel algorithmic techniques

i. speed-up
ii. recursive doubling, parallel prefix
iii. divide-and-conquer
iv. domain decomposition
v. data distribution (cf. P573)

(c) parallel algorithms

i. parallel sorting
ii. basic linear algebra operations (cf. P573)
iii. fast Fourier transforms (cf. P573)
iv. particle methods

As was the case in P573 we will attempt to cover various points of this syllabus by working on a couple of larger projects. Whatever science, mathematics, and computational techniques will be required in those should be explained in a sufficient detail.

1.4 What this Course is Not

I would like to reiterate that this is *not* a course in Numerical Analysis. Courses in Numerical Analysis are taught by the Department of Mathematics, see, for example, M471 (Numerical Analysis I), M472 (Numerical Analysis II), M568 (Time Series Analysis), M571 (Analysis of Numerical Methods I), M572 (Analysis of Numerical Methods II), M671 (Numerical Treatment of Differential and Integral Equations I), M672 (Numerical Treatment of Differential and Integral Equations II), M771 (Selected Topics in Numerical Analysis I), and M772 (Selected Topics in Numerical Analysis II). Neither is this a course in Computational Physics. This subject is taught by the Department of Physics, see, for example P410 (Computing Applications in Physics), P609 (Computational Physics), and P700 (Topics Course: Monte Carlo Methods in Physics). This is not a course in Computational Chemistry, Geophysics, Optics and Engineering either. All those topics are covered by respective schools.

Syllabus items covered by other courses will not be covered by this one

Selected topics of the Syllabus that are covered by those courses, will not be covered either by this course, or by P573 – unless they're needed to illustrate a particular point. See, for example, our discussion of Spectral Method and incomplete Euler gamma functions in P573.

1.5 Recommended Reading

Lecture Notes

As was the case with P573 I will provide you with rather detailed lecture notes in HTML and PostScript. These will be available on-line under

- `http://beige.ucs.indiana.edu/B673` (HTML version), and
- `http://beige.ucs.indiana.edu/gustav/B673.ps.gz` (PostScript version), or
- `/afs/ovpit.indiana.edu/common/www/htdocs/gustav/B673.ps.gz` (same).

The latter is the AFS address. There is also a pointer at

`http://www.cs.indiana.edu/dept/acad/courses.html`

towards the end of the page, which will send you directly to the lecture notes.

But you should also remember that these are just notes and not a book. They're likely to be sometimes incomplete or messy or even downright incorrect, since we're all prone to make mistakes.

The notes may well grow to a considerable size by the time the course is finished. Use GNU `ghostview` to view the PostScript document and select *new* pages for printing as they become available.

The reading list

The following is a somewhat incomplete list of books and other publications that I am going to base this course, B673, on. You should also check the recommended reading list for P573.

- "Designing and Building Parallel Programs, Concepts and Tools for Parallel Software Engineering" by I. Foster, Addison Wesley Publishing Company, 1994, ISBN 0-201-57594-9, pp. 380:

  A good broad review of parallel computing with numerous examples and case studies. Not a book from which to learn HPF or MPI programming though. But it gives a fairly good coverage of the issues.

- "Using MPI, Portable Parallel Programming with the Message-Passing Interface" by W. Gropp, E. Lusk, and A. Skjelum, The MIT Press, 1994, ISBN 0-262-57104-8, pp. 328:

An easy to follow introduction to Message Passing Interface, MPI, with numerous examples in C and F77

- MPI: A Message-Passing Interface Standard (in HTML)
- MPI: A Message-Passing Interface Standard (in PostScript)
- MPI-2: Extensions to the Message-Passing Interface (in PostScript)

  These three documents are available on-line from

  `http://beige.ucs.indiana.edu`

  and from

  `http://www.mcs.anl.gov`

  They define MPI. They are much more readable than one could expect, and abound in numerous examples.

- "Numerical Recipes, The Art of Scientific Computing" by W. H. Press, B. P. Flannery, S. A. Teukolsky and W. T. Vetterling, Cambridge University Press, 1986, ISBN 0 521 30811 9, pp. 818:

  This is one of the best and the most useful books for scientists working with computers, be it to analyse their experimental results, or develop numerical models. Written by four consummate practitioners of computational science with extensive academic and industrial experience, the book is positively hated by great many numerical analysts, primarily for not having mentioned their latest favourite method and papers. A good enough reason to buy it: stick to the classics. This book will serve you well for years to come. The ISBN number quoted above refers to one of its first editions. Since then the book has been reprinted and improved many times and in many ways, much like the Bible. Go for the latest edition.

- "Numerical Recipes in Fortran 90" by William Press, Saul Teukolsky, William Vetterling, and Brian Flannery, Cambridge University Press, 1996, pp. 550:

  As volume 2 of the Fortran Numerical Recipes series, this book takes up where volume 1 (see 1.5 above) leaves off. Volume 2 begins with three completely new chapters that provide a detailed introduction to the Fortran 90 language and then present the basic concepts of parallel programming, all with the same clarity and good cheer for which Numerical Recipes *is famous.*

- "Introduction to High-Performance Scientific Computing" by Lloyd D. Fosdick, Elizabeth R. Jessup, Carolyn J. C. Schauble, and Gitta Domik, The MIT Press, 1996, ISBN 0-262-06181-3, 750 pp., $60.00 (cloth):

> *This text evolved from a course given to undergraduate science and engineering majors at MIT. It covers most of our syllabus and we are going to use it quite frequently, although not all the time. The book is not too expensive, given its size and scope. It is a highly recommended reading for this course.*

- "Numerical Linear Algebra for High-Performance Computers" by J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst, Society for Industrial and Applied Mathematics, 1998, ISBN 0-89871-428-1, pp. 342:

  > *The book presents a unified treatment of recently developed techniques and current understanding about solving systems of linear equations and large-scale eigenvalue problems on high-performance computers. The book provides an introduction to the world of vector and parallel processing for these linear algebra applications.*

- "Solving Linear Systems on Vector and Shared Memory Computers" by J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst, Second Printing, Society for Industrial and Applied Mathematics, 1993, ISBN 0-89871-270-X, pp. 256:

  > *An older version of "Numerical Linear Algebra for High-Performance Computers".*

- "Introduction to Scientific Computing, A Matrix-Vector Approach Using MATLAB" by Charles F. van Loan, Prentice Hall, 1997, ISBN 0-13-125444-8, pp. 374:

  > *A good easy going introduction to* MATLAB *and to Octave. The latter is not mentioned in the book, but as you begin working with Octave you'll notice that it's uncannily similar to* MATLAB *(though* free*). The book covers some elementary numerical analysis too – not a bad way to learn about it.*

- "GNU Octave, A high-level interactive language for numerical computations", by John W. Eaton, Edition 3 for Octave version 2.0.13, February 1997. This text can be also found in

  ```
  /afs/ovpit.indiana.edu/common/gnu/doc/octave-2.0.13/interpreter
  ```

  and in Emacs info directory on our AFS cell:

  ```
  /afs/ovpit.indiana.edu/@sys/gnu/info
  ```

  > *An official Octave manual that is distributed together with Octave source. It covers a Brief Introduction, a Getting Started tutorial, and then goes into the details of the system discussing every available function.*

- "Explorations in Quantum Computing" by Colin P. Williams and Scott H. Clearwater, Springer Verlag, ISBN 0-387-94768-X, 1997, pp. 307:

  > *A very interesting and useful introduction to this difficult but at the same time so very promising field. Quantum computing may provide performance many orders of magnitude better than the best that you will* ever *be able to squeeze out of conventional computers based on* slooowly *diffusing semi-classical electrons trapped in a crystal lattice of even the fastest semiconductors. Quantum computing is also going to be orders of magnitude cheaper. You can do it even today with a cup of coffe (seriously) and a Nuclear Magnetic Resonance machine.*

1.6 Required Computer Accounts

You will need to obtain computer accounts on the Ships Cluster in the Student Building, and on the IBM SP. You will be also provided with AFS accounts. In order to obtain the Ships and the SP accounts you must have an IU Network ID. Connect to

`http://www.indiana.edu/ ucshelp/accounts.html`

for more information and on-line registration. Once you have the Network ID, use it to create required accounts. In case of any problems contact me first, and, failing that,

SP problems Mary Papakhian, `mpapakhi@indiana.edu`, phone: 855-2597

Ships problems Jeff Gronek, `jgronek@indiana.edu`, phone: 855-4937

1.7 Contact Details

How to find information about me and locate my office

Feel free to contact me any time you wish. See

`http://beige.ucs.indiana.edu/gustav`

for contact details. If you have a tool that can read Common Desktop Environment calendars, you will find my calendar at

`gustav@beige.ucs.indiana.edu`

1.8 The Bulletin Board

- There will be no lecture on the 18th of January, Martin Luther King, Jr. Day (Monday)
- There will be no lectures on the 15th (Monday) and 17th (Wednesday) of March, Spring Break and St. Patrick's Day

Chapter 2

Spectral Methods

2.1 Discrete Fourier Transform

Consider the following integral:

$$F(k) = \int_0^{2\pi} f(x) e^{-ikx}\,\mathrm{d}x \tag{2.1}$$

Assume that function $f(x)$ is sampled within the interval $[0, 2\pi]$ on a regularly spaced set of points x_l, $l = 0 \ldots n-1$. In that case we can approximate integral (2.1) by evaluating a finite sum

$$\begin{aligned} F_k &= \alpha \sum_{l=0}^{n-1} f(x_l) e^{-ik(2\pi/n)l} \\ &= \alpha \Big(f(x_0) e^{-ik(2\pi/n)0} + f(x_1) e^{-ik(2\pi/n)1} + f(x_2) e^{-ik(2\pi/n)2} + \cdots \\ &\quad + f(x_{n-1}) e^{-ik(2\pi/n)(n-1)} \Big) \end{aligned} \tag{2.2}$$

If every point x_l sits at the left hand side of its interval of length $\Delta x = 2\pi/n$, then the above evaluates to:

$$\begin{aligned} F_k = \alpha \Big(& f(x_0) e^{-ikx_0} + f(x_1) e^{-ikx_1} + f(x_2) e^{-ikx_2} + \cdots \\ & + f(x_{n-1}) e^{-ikx_{n-1}} \Big) \end{aligned} \tag{2.3}$$

and for $\alpha = \Delta x = 2\pi/n$ we get that

$$F_k \approx F(k) \tag{2.4}$$

Let us substitute $\omega_n = e^{-2\pi i/n}$ in equation (2.2):

$$F_k = \alpha \sum_{l=0}^{n-1} \omega_n^{kl} f(x_l) \tag{2.5}$$

We can rewrite this expression in the form of a matrix multiplication:

$$\begin{pmatrix} F_0 \\ F_1 \\ F_2 \\ \vdots \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n^{1\cdot 1} & \omega_n^{1\cdot 2} & \cdots & \omega_n^{1\cdot(n-1)} \\ 1 & \omega_n^{2\cdot 1} & \omega_n^{2\cdot 2} & \cdots & \omega_n^{2\cdot(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{(n-1)\cdot 1} & \omega_n^{(n-1)\cdot 2} & \cdots & \omega_n^{(n-1)\cdot(n-1)} \end{pmatrix} \begin{pmatrix} f(x_0) \\ f(x_1) \\ f(x_2) \\ \vdots \\ f(x_{n-1}) \end{pmatrix} \tag{2.6}$$

The discrete Fourier transform matrix shown in equation (2.6) is a complex Vandermonde matrix. Such matrices appear in polynomial interpolation. It is easy to write a simple Fortran program that builds a discrete Fourier transform matrix. But instead of doing it with Fortran, this time we'll use our interactive Fortran, Octave, to build a 4×4 discrete Fourier transform matrix.

```
Octave, version 2.0.13 (sparc-sun-solaris2.6).
Copyright (C) 1996, 1997, 1998 John W. Eaton.
This is free software with ABSOLUTELY NO WARRANTY.
For details, type `warranty'.

octave:1> w = exp(-2 * pi * i / 4)
w = -i
octave:2> F = ones(4,4)
F =

  1  1  1  1
  1  1  1  1
  1  1  1  1
  1  1  1  1

octave:3> (0:3)
ans =

  0  1  2  3

octave:4> F(2,:) = w .^ (0:3)
F =

  1  1  1  1
  1 -i -1  i
  1  1  1  1
  1  1  1  1

octave:5> F(3,:) = w .^ (2 * (0:3))
F =

  1  1  1  1
  1 -i -1  i
  1 -1  1 -1
  1  1  1  1

octave:6> F(4,:) = w .^ (3 * (0:3))
F =

  1  1  1  1
  1 -i -1  i
  1 -1  1 -1
  1  i -1 -i

octave:7>
```

Another way to accomplish the same would be as follows:

```
octave:8> F = ones(4,4)
F =

  1  1  1  1
  1  1  1  1
  1  1  1  1
  1  1  1  1

octave:9> F(2,:) = w .^ (0:3)
F =

  1  1  1  1
  1 -i -1  i
  1  1  1  1
  1  1  1  1

octave:10> F(3,:) = F(2,:) .* F(2,:)
F =

  1  1  1  1
  1 -i -1  i
  1 -1  1 -1
  1  1  1  1

octave:11> F(4,:) = F(2,:) .* F(3,:)
F =

  1  1  1  1
  1 -i -1  i
  1 -1  1 -1
  1  i -1 -i

octave:13>
```

This is a faster way because multiplication is cheaper than exponentiation.

Once we have built the discrete Fourier transform matrix we can attempt to find a numerical Fourier transform of, say, a sine function:

```
octave:39> x = [ 0, 2*pi/4, 2*2*pi/4, 3*2*pi/4 ]
x =

  0.00000  1.57080  3.14159  4.71239

octave:40> x = sin(x)
x =

   0.00000   1.00000   0.00000  -1.00000

octave:41> 2 * pi / 4 * F * x'
```

```
ans =

   0.00000 + 0.00000i
   0.00000 - 3.14159i
   0.00000 + 0.00000i
  -0.00000 + 3.14159i

octave:42>
```

Our numerical result, neglecting the last harmonic, is that

$$\int_0^{2\pi} \sin x \, e^{-ikx} \, \mathrm{d}x = -\pi i \delta(k-1)$$

which is just right, because

$$\begin{aligned}\int_0^{2\pi} \sin x \, e^{-ikx} \, \mathrm{d}x &= \int_0^{2\pi} \sin x \; (\cos(kx) - i\sin(kx)) \; \mathrm{d}x \\ = -i\delta(k-1) \int_0^{2\pi} &\sin x \sin x \, \mathrm{d}x = -i\delta(k-1)\pi\end{aligned}$$

Recall that you can check that integration easily with Emacs calc, e.g.,

```
alg' integ(sin(x)^2, x, 0, 2 * pi)
     pi
```

but remember to do it in the radians mode.

What about that last harmonic? Repeating the same computation for $n = 8$ returns:

```
octave:43> w = exp(-2 * pi * i / 8)
w = 0.70711 - 0.70711i
octave:44> F = ones(8,8);
octave:45> for k = 2:8
> F(k,:) = w .^ ((k - 1) * (0:7));
> end
octave:46> x = 2 * pi / 8 * (0:7);
octave:47> x = sin(x);
octave:48> 2 * pi / 8 * F * x'
ans =

  -0.00000 + 0.00000i
   0.00000 - 3.14159i
   0.00000 + 0.00000i
   0.00000 + 0.00000i
   0.00000 + 0.00000i
   0.00000 + 0.00000i
   0.00000 + 0.00000i
  -0.00000 + 3.14159i

octave:49>
```

So we are still getting the same result, $-i\pi\delta(k-1)$, and the only difference is that the last harmonic has been moved further away.

Let us try a signal with higher harmonics:

```
octave:45> x = 2 * pi / 8 * (0:7);
octave:46> y = sin(x) + 0.5 * sin (2 * x);
octave:47> 2 * pi / 8 * F * y'
ans =

   0.00000 + 0.00000i
   0.00000 - 3.14159i
   0.00000 - 1.57080i
  -0.00000 + 0.00000i
   0.00000 + 0.00000i
   0.00000 + 0.00000i
  -0.00000 + 1.57080i
  -0.00000 + 3.14159i

octave:48> y = cos(x) + sin(2 * x) - cos(3 * x);
octave:49> 2 * pi / 8 * F * y'
ans =

   0.00000 + 0.00000i
   3.14159 + 0.00000i
   0.00000 - 3.14159i
  -3.14159 - 0.00000i
  -0.00000 + 0.00000i
  -3.14159 - 0.00000i
  -0.00000 + 3.14159i
   3.14159 + 0.00000i

octave:50>
```

What is happening is that harmonics are inserted from both ends. The reason for this is that the discrete Fourier transform returns amplitudes that correspond to both positive and negative wave numbers. Amplitudes for positive wave numbers are returned in slots labeled with $l = 1 \ldots n/2 - 1$. Amplitudes for negative wave numbers are returned in reverse order in slots $l = n/2+1 \ldots n-1$ and the Nyquist freqency amplitude

$$f_c = \frac{1}{2\Delta} \tag{2.7}$$

or

$$k_c = 2\pi f_c = \frac{2\pi}{2\Delta}, \tag{2.8}$$

where Δ is the sampling interval, is returned in slot $l = n/2$. The amplitude that corresponds to the zero frequency, i.e., a constant shift, is returned in slot $l = 0$. Because in Fortran arrays are normally numbered by default from 1 through n (but this can be changed with an appropriate declaration) you've got to add 1 to l in order to refer to an appropriate slot in a Fortran array.

In our case $\Delta = 2\pi/8$, $k_c = 4$, and the Nyquist frequency Amplitude is $-0.0 + 0.0i$.

How did those negative frequencies creep into our sum:

$$F_k = \alpha \sum_{l=0}^{n-1} f(x_l) e^{-ik(2\pi/n)l}$$

Observe that F_k is periodic in k with period equal to n. Therefore:

$$F_{n-k} = F_{-k} \tag{2.9}$$

In other words only half of the vector returned by a discrete Fourier transform corresponds to positive wave numbers. So, if you want, say, m amplitudes associated with positive wave numbers, you must sample your signal on at least $2m + 2$ points[1]. Conversely, if you sample on $2m + 2$ points the maximum frequency amplitude will correpond not to $\omega = 2m + 1$ but to $k_c = m + 1$.

What is going to happen if your signal contains harmonics that are higher than the Nyquist frequency k_c? It turns out that all of the power spectral density that lies outside of the range $[-k_c, +k_c]$ will be sneakily moved into that range, i.e., any amplitude that corresponds to a frequency outside that range is going to be *aliased* into that range.

There is no way to overcome *aliasing* other than to know the bandwidth limit of a signal in advance and then sample sufficiently densely, so that the Nyquist frequency is higher than the bandwidth limit.

2.2 Fast Fourier Transform

Fast Fourier Transform is a Discrete Fourier Transform calculated more efficiently. In order to evaluate the Discrete Fourier Transform Vandermonde matrix the way we have done it in the previous section we needed to evaluate somewhat less than $n \times n$ terms because the first row and the first column are all ones. But certain economies can be attempted. This derives from the observation that

$$\omega_n^k = e^{-2\pi i k/n}$$

is periodic in k and the period is n. This means that we don't really have to evaluate all powers between $\omega^{1\times 1}$ and $\omega^{(n-1)\times(n-1)}$, because they will keep repeating. For example the Fourier transform matrix for $n = 8$ would have the following terms:

$$^8\mathbf{\Lambda} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega_8^1 & \omega_8^2 & \omega_8^3 & \omega_8^4 & \omega_8^5 & \omega_8^6 & \omega_8^7 \\ 1 & \omega_8^2 & \omega_8^4 & \omega_8^6 & 1 & \omega_8^2 & \omega_8^4 & \omega_8^6 \\ 1 & \omega_8^3 & \omega_8^6 & \omega_8^1 & \omega_8^4 & \omega_8^7 & \omega_8^2 & \omega_8^5 \\ 1 & \omega_8^4 & 1 & \omega_8^4 & 1 & \omega_8^4 & 1 & \omega_8^4 \\ 1 & \omega_8^5 & \omega_8^2 & \omega_8^7 & \omega_8^4 & \omega_8^1 & \omega_8^6 & \omega_8^3 \\ 1 & \omega_8^6 & \omega_8^4 & \omega_8^2 & 1 & \omega_8^6 & \omega_8^4 & \omega_8^2 \\ 1 & \omega_8^7 & \omega_8^6 & \omega_8^5 & \omega_8^4 & \omega_8^3 & \omega_8^2 & \omega_8^1 \end{pmatrix} \tag{2.10}$$

It transpires that in order to evaluate a discrete Fourier Transform all that's needed is to evaluate $\omega_n^1 \dots \omega_n^{n-1}$. Then you just need to insert those terms

[1]It is for a good reason that the PESSL Spectral Method example we went through in our previous lecture, P573, asked for 7 harmonics in each direction, because $2 \times 7 + 2 = 16 = 2^4$, which is a good number for FFT.

cleverly in appropriate places in the Vandermonde matrix – also, observe that matrix (2.10) is symmetric. Now the insertion of $\omega_n^1 \dots \omega_n^{n-1}$ into appropriate spots is not going to be free: those places have to be computed one way or the other and data may have to be moved around too. But the resulting algorithm is not going to be $\mathcal{O}(n^2)$. It is going to be $\mathcal{O}(n \log n)$ instead.

Consider matrix given by (2.10). Let us rearrange the columns of the matrix and the entries of the corresponding vector f_k in the following way:

$$[1,2,3,4,5,6,7,8] \longrightarrow [1,3,5,7,2,4,6,8],$$

i.e., we put all the odd columns first and then the even ones. So now the matrix looks as follows:

$$\begin{pmatrix} {}^8F_0 \\ {}^8F_1 \\ {}^8F_2 \\ {}^8F_3 \\ {}^8F_4 \\ {}^8F_5 \\ {}^8F_6 \\ {}^8F_7 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega_8^2 & \omega_8^4 & \omega_8^6 & \omega_8^1 & \omega_8^3 & \omega_8^5 & \omega_8^7 \\ 1 & \omega_8^4 & 1 & \omega_8^4 & \omega_8^2 & \omega_8^6 & \omega_8^2 & \omega_8^6 \\ 1 & \omega_8^6 & \omega_8^4 & \omega_8^2 & \omega_8^3 & \omega_8^1 & \omega_8^7 & \omega_8^5 \\ 1 & 1 & 1 & 1 & \omega_8^4 & \omega_8^4 & \omega_8^4 & \omega_8^4 \\ 1 & \omega_8^2 & \omega_8^4 & \omega_8^6 & \omega_8^5 & \omega_8^7 & \omega_8^1 & \omega_8^3 \\ 1 & \omega_8^4 & 1 & \omega_8^4 & \omega_8^6 & \omega_8^2 & \omega_8^6 & \omega_8^2 \\ 1 & \omega_8^6 & \omega_8^4 & \omega_8^2 & \omega_8^7 & \omega_8^5 & \omega_8^3 & \omega_8^1 \end{pmatrix} \begin{pmatrix} f(x_0) \\ f(x_2) \\ f(x_4) \\ f(x_6) \\ f(x_1) \\ f(x_3) \\ f(x_5) \\ f(x_7) \end{pmatrix} \tag{2.11}$$

Now, observe that

$$\begin{aligned} \left(\omega_8^1, \omega_8^3, \omega_8^5, \omega_8^7\right) &= \omega_8^1 \cdot \left(1, \omega_8^2, \omega_8^4, \omega_8^6\right) \\ \left(\omega_8^2, \omega_8^6, \omega_8^2, \omega_8^6\right) &= \omega_8^2 \cdot \left(1, \omega_8^4, 1, \omega_8^4\right) \\ \left(\omega_8^3, \omega_8^1, \omega_8^7, \omega_8^5\right) &= \omega_8^3 \cdot \left(1, \omega_8^6, \omega_8^4, \omega_8^2\right) \end{aligned} \tag{2.12}$$

and that

$${}^4\mathbf{\Lambda} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega_8^2 & \omega_8^4 & \omega_8^6 \\ 1 & \omega_8^4 & 1 & \omega_8^4 \\ 1 & \omega_8^6 & \omega_8^4 & \omega_8^2 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega_4^1 & \omega_4^2 & \omega_4^3 \\ 1 & \omega_4^2 & 1 & \omega_4^2 \\ 1 & \omega_4^3 & \omega_4^2 & \omega_4^1 \end{pmatrix} \tag{2.13}$$

is a Vandermonde matrix for the 4-point discrete Fourier transform, and that multiplications in equation (2.12) can be expressed as:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \omega_8 & 0 & 0 \\ 0 & 0 & \omega_8^2 & 0 \\ 0 & 0 & 0 & \omega_8^3 \end{pmatrix} \cdot {}^4\mathbf{\Lambda} = {}^8\boldsymbol{D}_4 \cdot {}^4\mathbf{\Lambda} \tag{2.14}$$

Last, observe that the lower right corner of ${}^8\mathbf{\Lambda}_{[1,3,5,7,2,4,6,8]}$ is simply $-{}^8\boldsymbol{D}_4 \cdot {}^4\mathbf{\Lambda}$ because

$$\begin{pmatrix} \omega_8^4 & \omega_8^4 & \omega_8^4 & \omega_8^4 \\ \omega_8^5 & \omega_8^7 & \omega_8^1 & \omega_8^3 \\ \omega_8^6 & \omega_8^2 & \omega_8^6 & \omega_8^2 \\ \omega_8^7 & \omega_8^5 & \omega_8^3 & \omega_8^1 \end{pmatrix} = \omega_8^4 \begin{pmatrix} 1 & 1 & 1 & 1 \\ \omega_8^1 & \omega_8^3 & \omega_8^5 & \omega_8^7 \\ \omega_8^2 & \omega_8^6 & \omega_8^2 & \omega_8^6 \\ \omega_8^3 & \omega_8^1 & \omega_8^7 & \omega_8^5 \end{pmatrix} \tag{2.15}$$

and $\omega_8^4 = e^{-2\pi i 4/8} = e^{-\pi i} = -1$.

In summary equation (2.11) can be rewritten as follows:

$$\begin{aligned}
{}^8\boldsymbol{F}(\boldsymbol{f}_{[1:8]}) &= \begin{pmatrix} {}^4\boldsymbol{\Lambda}, & {}^8\boldsymbol{D}_4 \cdot {}^4\boldsymbol{\Lambda} \\ {}^4\boldsymbol{\Lambda}, & -{}^8\boldsymbol{D}_4 \cdot {}^4\boldsymbol{\Lambda} \end{pmatrix} \begin{pmatrix} \boldsymbol{f}_{[1:8:2]} \\ \boldsymbol{f}_{[2:8:2]} \end{pmatrix} \\
&= \begin{pmatrix} \mathbf{1} & {}^8\boldsymbol{D}_4 \\ \mathbf{1} & -{}^8\boldsymbol{D}_4 \end{pmatrix} \cdot \begin{pmatrix} {}^4\boldsymbol{\Lambda} \cdot \boldsymbol{f}_{[1:8:2]} \\ {}^4\boldsymbol{\Lambda} \cdot \boldsymbol{f}_{[2:8:2]} \end{pmatrix} \\
&= \begin{pmatrix} \mathbf{1} & {}^8\boldsymbol{D}_4 \\ \mathbf{1} & -{}^8\boldsymbol{D}_4 \end{pmatrix} \cdot \begin{pmatrix} {}^4\boldsymbol{F}(\boldsymbol{f}_{[1:8:2]}) \\ {}^4\boldsymbol{F}(\boldsymbol{f}_{[2:8:2]}) \end{pmatrix}
\end{aligned} \tag{2.16}$$

Applying the same reasoning to ${}^4\boldsymbol{F}(\boldsymbol{f}_{[1:8:2]})$ and to ${}^4\boldsymbol{F}(\boldsymbol{f}_{[2:8:2]})$ we can write without much further ado:

$$\begin{aligned}
{}^4\boldsymbol{F}(\boldsymbol{f}_{[1:8:2]}) &= \begin{pmatrix} \mathbf{1} & {}^4\boldsymbol{D}_2 \\ \mathbf{1} & -{}^4\boldsymbol{D}_2 \end{pmatrix} \cdot \begin{pmatrix} {}^2\boldsymbol{F}(\boldsymbol{f}_{[1:8:4]}) \\ {}^2\boldsymbol{F}(\boldsymbol{f}_{[3:8:4]}) \end{pmatrix} \\
{}^4\boldsymbol{F}(\boldsymbol{f}_{[2:8:2]}) &= \begin{pmatrix} \mathbf{1} & {}^4\boldsymbol{D}_2 \\ \mathbf{1} & -{}^4\boldsymbol{D}_2 \end{pmatrix} \cdot \begin{pmatrix} {}^2\boldsymbol{F}(\boldsymbol{f}_{[2:8:4]}) \\ {}^2\boldsymbol{F}(\boldsymbol{f}_{[4:8:4]}) \end{pmatrix}
\end{aligned} \tag{2.17}$$

And we repeat it once more to get:

$$\begin{aligned}
{}^2\boldsymbol{F}(\boldsymbol{f}_{[1:8:4]}) &= \begin{pmatrix} \mathbf{1} & {}^2\boldsymbol{D}_1 \\ \mathbf{1} & -{}^2\boldsymbol{D}_1 \end{pmatrix} \cdot \begin{pmatrix} {}^1\boldsymbol{F}(\boldsymbol{f}_1) \\ {}^1\boldsymbol{F}(\boldsymbol{f}_5) \end{pmatrix} = \begin{pmatrix} \mathbf{1} & {}^2\boldsymbol{D}_1 \\ \mathbf{1} & -{}^2\boldsymbol{D}_1 \end{pmatrix} \cdot \begin{pmatrix} f(x_0) \\ f(x_4) \end{pmatrix} \\
{}^2\boldsymbol{F}(\boldsymbol{f}_{[3:8:4]}) &= \begin{pmatrix} \mathbf{1} & {}^2\boldsymbol{D}_1 \\ \mathbf{1} & -{}^2\boldsymbol{D}_1 \end{pmatrix} \cdot \begin{pmatrix} {}^1\boldsymbol{F}(\boldsymbol{f}_3) \\ {}^1\boldsymbol{F}(\boldsymbol{f}_7) \end{pmatrix} = \begin{pmatrix} \mathbf{1} & {}^2\boldsymbol{D}_1 \\ \mathbf{1} & -{}^2\boldsymbol{D}_1 \end{pmatrix} \cdot \begin{pmatrix} f(x_2) \\ f(x_6) \end{pmatrix} \\
{}^2\boldsymbol{F}(\boldsymbol{f}_{[2:8:4]}) &= \begin{pmatrix} \mathbf{1} & {}^2\boldsymbol{D}_1 \\ \mathbf{1} & -{}^2\boldsymbol{D}_1 \end{pmatrix} \cdot \begin{pmatrix} {}^1\boldsymbol{F}(\boldsymbol{f}_2) \\ {}^1\boldsymbol{F}(\boldsymbol{f}_6) \end{pmatrix} = \begin{pmatrix} \mathbf{1} & {}^2\boldsymbol{D}_1 \\ \mathbf{1} & -{}^2\boldsymbol{D}_1 \end{pmatrix} \cdot \begin{pmatrix} f(x_1) \\ f(x_5) \end{pmatrix} \\
{}^2\boldsymbol{F}(\boldsymbol{f}_{[4:8:4]}) &= \begin{pmatrix} \mathbf{1} & {}^2\boldsymbol{D}_1 \\ \mathbf{1} & -{}^2\boldsymbol{D}_1 \end{pmatrix} \cdot \begin{pmatrix} {}^1\boldsymbol{F}(\boldsymbol{f}_4) \\ {}^1\boldsymbol{F}(\boldsymbol{f}_8) \end{pmatrix} = \begin{pmatrix} \mathbf{1} & {}^2\boldsymbol{D}_1 \\ \mathbf{1} & -{}^2\boldsymbol{D}_1 \end{pmatrix} \cdot \begin{pmatrix} f(x_3) \\ f(x_7) \end{pmatrix}
\end{aligned} \tag{2.18}$$

where I have made use of the fact that ${}^1\boldsymbol{F}(\boldsymbol{f}_k) = \boldsymbol{f}_k$.

And this is Fast Fourier Transform.

When it comes to coding this algorithm, you can either start from the top and then write a recursive procedure, or you can start at the bottom and evaluate four 2-point transforms first, then two 4-point transforms and finally the single 8-point transform.

Observe that you never really evaluate and store the full Vandermonde discrete Fourier tranform matrix. Matrices ${}^{2k}\boldsymbol{D}_k$ are diagonal, so they can be stored simply as vectors and the matrix operation that converts two n-point transforms into one $2n$-point transform can be implemented as two simple equations. The only ω_n^k terms that are evaluated while calculating ${}^{2k}\boldsymbol{D}_k$, are the ones that are really needed. Periodicity of ω_n^k is automatically taken care of.

2.2.1 A Recursive Implementation of the Fast Fourier Transform

Before we can embark on implementing the algorithm discussed above in Octave one word of caution. Throughout this course I have been referring to Octave as "interactive Fortran", but, although in many respects Octave's notation is quite similar to that of Fortran, sometimes it is annoyingly different. The situation is a little like with differences between Bourne Shell and C-shell - those little differences can be quite enough to derail one's shell scripts and one's composure too.

And the difference that I need to point to you right now relates to the way ranges and strides are described in both environments. And so, in Fortran `(1:10:2)` means "take integers from 1 through 10 with a stride of 2", i.e., 1, 3, 5, 7, 9. In Octave the same is accomplished by `(1:2:10)`:

```
octave:5> (1:2:10)
ans =

  1  3  5  7  9

octave:6>
```

Now let us have a look at the top FFT formula:

$$ {}^{8}\boldsymbol{F}(\boldsymbol{f}_{[1:8]}) = \begin{pmatrix} \mathbf{1} & {}^{8}\boldsymbol{D}_4 \\ \mathbf{1} & -{}^{8}\boldsymbol{D}_4 \end{pmatrix} \cdot \begin{pmatrix} {}^{4}\boldsymbol{F}(\boldsymbol{f}_{[1:8:2]}) \\ {}^{4}\boldsymbol{F}(\boldsymbol{f}_{[2:8:2]}) \end{pmatrix} \tag{2.19} $$

How would we implement this in Octave? First let us assume that our input data lives on an 8-entries long vector `x`, and that we have a function called `my_fft` that can evaluate a discrete Fourier Transform from any regularly spaced subset of `x`, for example `x(1:2:8)` – remember that here we already use Octave notation, which means that the second, not the third, integer in the sequence `(1:2:8)` defines the stride. So

$$ {}^{4}\boldsymbol{F}(\boldsymbol{f}_{[1:8:2]}) = \texttt{my\_fft}\,(x(1:2:8)) \tag{2.20} $$

and

$$ {}^{4}\boldsymbol{F}(\boldsymbol{f}_{[2:8:2]}) = \texttt{my\_fft}\,(x(2:2:8)) \tag{2.21} $$

What the function is going to deliver in this case is an array of 4 complex numbers that represents a 4-point discrete Fourier transform. To get the 8-point transform:

$$ \begin{aligned} {}^{8}\boldsymbol{F}_{(1:4)} &= \texttt{my\_fft}\,(x(1:2:8)) + [1, \omega_8, \omega_8^2, \omega_8^3] \cdot \texttt{my\_fft}\,(x(2:2:8)) \\ {}^{8}\boldsymbol{F}_{(5:8)} &= \texttt{my\_fft}\,(x(1:2:8)) - [1, \omega_8, \omega_8^2, \omega_8^3] \cdot \texttt{my\_fft}\,(x(2:2:8)) \end{aligned} \tag{2.22} $$

And the beauty of recursive algorithms that this is really the only thing that we have to code. Plus the condition that stops the recursion, of course.

So, here is the Octave code:

```
octave:1> function y = my_fft(x)
> n = length(x);
> if n == 1
>    y = x;
> else
>    m = n/2;
>    y_top = my_fft(x(1:2:n));
>    y_bottom = my_fft(x(2:2:n));
>    d = exp(-2 * pi * i / n) .^ (0:m-1);
>    z = d .* y_bottom;
>    y = [ y_top + z , y_top - z ];
> end
> endfunction
octave:2> x = 2 * pi / 8 * (0:7);
octave:3> x = sin(x);
octave:4> 2 * pi / 8 * my_fft(x)
ans =

 Columns 1 through 3:

   0.00000 + 0.00000i  -0.00000 - 3.14159i   0.00000 - 0.00000i

 Columns 4 through 6:

   0.00000 + 0.00000i   0.00000 + 0.00000i   0.00000 + 0.00000i

 Columns 7 and 8:

   0.00000 + 0.00000i  -0.00000 + 3.14159i

octave:35>
```

It is easy enough to translate this code to Fortran-90, remembering, of course, that `(1:2:n)` needs to be replaced with `(1:n:2)`. Here is the translation not only of function `my_fft` itself, which in the Fortran program shown below is implemented as a `subroutine fft`, but also of computations that lead to setting up vector `x`:

```
PROGRAM try_fft

  IMPLICIT NONE

  DOUBLE PRECISION, PARAMETER :: pi = 3.141592653589793d0
  INTEGER :: i
  COMPLEX(kind=8), DIMENSION(8) :: x, y

  x = (/ (i, i=0,7) /)
  x = 2 * pi / 8 * x
  x = SIN(x)
  CALL fft(x, y); y = 2 * pi / 8 * y
  WRITE(*,'(8 ("(", en12.3, ",", en12.3, ")", /))') y
  STOP 0

CONTAINS

  RECURSIVE SUBROUTINE fft(x, y)
```

```
    IMPLICIT NONE

    COMPLEX(kind=8), DIMENSION(:) :: x
    COMPLEX(kind=8), DIMENSION(SIZE(x)) :: y

    COMPLEX(kind=8), DIMENSION(:), ALLOCATABLE :: y_top, y_bottom, z, d
    DOUBLE PRECISION, PARAMETER :: pi = 3.141592653589793d0

    INTEGER :: n, m, i

    n = SIZE(x)
    IF (n .EQ. 1) THEN
       y = x
    ELSE
       m = n/2
       ALLOCATE(y_top(m), y_bottom(m), d(m), z(m))
       CALL fft(x(1:n:2), y_top)
       CALL fft(x(2:n:2), y_bottom)
       d = (/ (EXP(-2. * pi * (0, 1.) / n) ** i, i = 0, m-1) /)
       z = d * y_bottom
       y(1:m) = y_top + z; y(m+1:n) = y_top - z
    END IF
  END SUBROUTINE fft

END PROGRAM try_fft
```

There isn't really much more writing here than in the Octave program, although we have to be more formal, and, in particular, all variables must be declared, especially following the

```
IMPLICIT NONE
```

statement, and we have to remember to allocate space for `y_top`, `y_bottom`, `d` and `z`.

Here is how this program compiles and runs:

```
gustav@blanc:../src 18:01:22 !569 $ f90 -o fft fft.f90
gustav@blanc:../src 18:04:08 !570 $ ./fft
(   8.987E-18,   0.000E+00)
(-174.393E-18,  -3.142E+00)
(  96.184E-18, -87.197E-18)
( 418.010E-18,   0.000E+00)
( 183.380E-18,   0.000E+00)
( 174.393E-18,   0.000E+00)
(  96.184E-18,  87.197E-18)
(-802.744E-18,   3.142E+00)

 STOP 0
gustav@blanc:../src 18:04:11 !571 $
```

The algorithm presented in this section is perhaps the simplest of all FFT algorithms. It suffers from two inefficiencies. The first one is the statement:

```
d = (/ (EXP(-2. * pi * (0, 1.) / n) ** i, i = 0, m-1) /)
```

which results in calculating more terms of ω_n^k than is really necessary: some will be unnecessarily recalculated on climbing up from the recursion. This can be addressed relatively trivially: for example we could precompute only as many ω_n^k as are needed and then we would simply look them up.

The other inefficiency is recursion itself, which although it can be made nearly as efficient as iterations in some cases (e.g., for the so called "tail recursion"), it will usually take more resources and run slower than a simple `DO` loop.

Regarding Fortran itself there are two new elements in the code, which you haven't encountered yet. The first one is the so called *engineering notation*, which is used when formatting the complex numbers:

```
  WRITE(*,'(8 ("(", en12.3, ",", en12.3, ")", /))') y
```

The format is flagged by `en`. The other two formats that can be used for floating point numbers are `e` and `f`.

The other novelty is the keyword `RECURSIVE` that appears in front of `SUBROUTINE fft`. A subroutine or a function must be declared `RECURSIVE` if it is such. Fortran-77 and earlier Fortrans did not support recursion, so this is still a new and seldom used feature to most Fortran programmers.

2.2.2 The Danielson Lanczos Algorithm

The algorithm presented above can be rewritten in a more general way as follows:

$$\begin{aligned}
{}^{n}F_k &= \sum_{j=0}^{n-1} e^{-2\pi ikj/n} f_j \\
&= \sum_{j=0}^{n/2-1} e^{-2\pi ik(2j)/n} f_{2j} + \sum_{j=0}^{n/2-1} e^{-2\pi ik(2j+1)/n} f_{2j+1} \\
&= \sum_{j=0}^{n/2-1} e^{-2\pi ikj/(n/2)} f_{2j} + \omega_n^k \sum_{j=0}^{n/2-1} e^{-2\pi ikj/(n/2)} f_{2j+1} \\
&= {}^{n/2}F_k^e + \omega_n^k \cdot {}^{n/2}F_k^o \qquad (2.23)
\end{aligned}$$

where ${}^{n/2}F_k^e$ is an $n/2$-point DFT based on sampling *even* points of the original DFT and evaluated at harmonic k (observe that here we may have that $k > n/2-1$, but since $e^{-2\pi ik/(n/2)}$ is periodic in k with period $n/2$, we can always reduce k to something that's within the interval $[0, n/2-1]$), and ${}^{n/2}F_k^o$ is an $n/2$-point DFT based on sampling *odd* points of the original DFT and evaluated at harmonic k.

At this stage we can apply the same formula recursively to ${}^{n/2}F_k^e$ and to ${}^{n/2}F_k^o$ and reduce the problem to the evaluation of ${}^{n/4}F_k^{ee}$, ${}^{n/4}F_k^{eo}$, ${}^{n/4}F_k^{oe}$, ${}^{n/4}F_k^{oo}$. This, in turn, can be reduced to the evaluation of ${}^{n/8}F_k^{eee}$, ${}^{n/8}F_k^{eeo}$, and so on, until ${}^{n/8}F_k^{ooo}$ ($2^3 = 8$ terms).

At the end of this process we're going to hit:

$$ {}^{n/n}F_k^{oooeee\ldots eo} = f_m \tag{2.24} $$

where the m can be evaluated as follows:

- Reverse the pattern of es and os back to front.
- Replace every e with 0 and every o with 1.
- Reinterpret the obtained stream of 1s and 0s as a binary representation of a number. That number is our m.

Let us go back to our original example, where we had a look at ${}^8\boldsymbol{F}$ and evaluate, say, ${}^8F_5$. According to our formula:

$$\begin{aligned}
{}^8F_5 &= {}^4F_5^e + \omega_8^5 \cdot {}^4F_5^o \\
&= \left({}^2F_5^{ee} + \omega_4^5 \cdot {}^2F_5^{eo}\right) + \omega_8^5 \cdot \left({}^2F_5^{oe} + \omega_4^5 \cdot {}^2F_5^{oo}\right) \\
&= \left(\left({}^1F_5^{eee} + \omega_2^5 \cdot {}^1F_5^{eeo}\right) + \omega_4^5 \cdot \left({}^1F_5^{eoe} + \omega_2^5 \cdot {}^1F_5^{eoo}\right)\right) \\
&\quad + \omega_8^5 \cdot \left(\left({}^1F_5^{oee} + \omega_2^5 \cdot {}^1F_5^{oeo}\right) + \omega_4^5 \cdot \left({}^1F_5^{ooe} + \omega_2^5 \cdot {}^1F_5^{ooo}\right)\right) \\
&= \left(\left(f_{eee} + \omega_2^5 \cdot f_{eeo}\right) + \omega_4^5 \cdot \left(f_{eoe} + \omega_2^5 \cdot f_{eoo}\right)\right) \\
&\quad + \omega_8^5 \cdot \left(\left(f_{oee} + \omega_2^5 \cdot f_{oeo}\right) + \omega_4^5 \cdot \left(f_{ooe} + \omega_2^5 \cdot f_{ooo}\right)\right) \\
&= \left(\left(f_{000} + \omega_2^5 \cdot f_{100}\right) + \omega_4^5 \cdot \left(f_{010} + \omega_2^5 \cdot f_{110}\right)\right) \\
&\quad + \omega_8^5 \cdot \left(\left(f_{001} + \omega_2^5 \cdot f_{101}\right) + \omega_4^5 \cdot \left(f_{011} + \omega_2^5 \cdot f_{111}\right)\right) \\
&= \left(\left(f_0 + \omega_2^5 \cdot f_4\right) + \omega_4^5 \cdot \left(f_2 + \omega_2^5 \cdot f_6\right)\right) \\
&\quad + \omega_8^5 \cdot \left(\left(f_1 + \omega_2^5 \cdot f_5\right) + \omega_4^5 \cdot \left(f_3 + \omega_2^5 \cdot f_7\right)\right)
\end{aligned}$$

Observe that reversing the bits that correspond to numbers $1, 2, 3, \ldots, n$ back to front will automatically generate the required pairs:

$$\begin{array}{ccccccc}
0 & \to & 000 & \to 000 & \to & 0 \\
1 & \to & 001 & \to 100 & \to & 4 \\
2 & \to & 010 & \to 010 & \to & 2 \\
3 & \to & 011 & \to 110 & \to & 6 \\
4 & \to & 100 & \to 001 & \to & 1 \\
5 & \to & 101 & \to 101 & \to & 5 \\
6 & \to & 110 & \to 011 & \to & 3 \\
7 & \to & 111 & \to 111 & \to & 7
\end{array}$$

And this suggests the following computational strategy:

1. Sort the data into bit-reversed order.
2. Evaluate 4 ${}^2\boldsymbol{F}$ transforms using pairs that result from that ordering.
3. Evaluate 2 ${}^4\boldsymbol{F}$ transforms using ordering obtained in the previous step.

4. Evaluate 1 $^8\boldsymbol{F}$ transform using ordering obtained in the previous step.

How to sort the data into bit-reversed order?
There is a Fortran function that will come handy. The function is

```
ISHFTC(I, Shift, Length)
```

and it works as follows: `ISHFTC(I, 3, 17)` will shift circularly 17 rightmost bits of `I` by 3 places to the left. If the `Shift` parameter is negative then the bits are shifted to the right.

How is this function going to help? Observe how we can bit-reverse a sequence by applying a series of righmost circular shifts with diminishing length to it:

$$\begin{array}{lcl}
 & & [a,b,c,d,e,f,g,h] \\
\texttt{ishftc(i, -1, 8)} & \to & [h,a,b,c,d,e,f,g] \\
\texttt{ishftc(i, -1, 7)} & \to & [h,g,a,b,c,d,e,f] \\
\texttt{ishftc(i, -1, 6)} & \to & [h,g,f,a,b,c,d,e] \\
\texttt{ishftc(i, -1, 5)} & \to & [h,g,f,e,a,b,c,d] \\
\texttt{ishftc(i, -1, 4)} & \to & [h,g,f,e,d,a,b,c] \\
\texttt{ishftc(i, -1, 3)} & \to & [h,g,f,e,d,c,a,b] \\
\texttt{ishftc(i, -1, 2)} & \to & [h,g,f,e,d,c,b,a]
\end{array}$$

Here is a Fortran code that does the same:

```
PROGRAM reverse

  INTEGER i

  DO i = 0, 7
     WRITE(*,*) 'i = ', i, '  bit_reverse(i, 3) = ', bit_reverse(i, 3)
  END DO

CONTAINS

  FUNCTION bit_reverse(i, size)
    INTEGER :: bit_reverse
    INTEGER, INTENT(in) :: i, size

    INTEGER :: length, temp

    temp = i
    DO length = size, 2, -1
       temp = ISHFTC(temp, -1, length)
    END DO
    bit_reverse = temp

  END FUNCTION bit_reverse

END PROGRAM reverse
```

And here is how it compiles and runs:

```
gustav@blanc:../src 20:25:45 !600 $ f90 -o reverse reverse.f90
gustav@blanc:../src 20:38:44 !601 $ ./reverse
 i = 0  bit_reverse(i, 3) = 0
 i = 1  bit_reverse(i, 3) = 4
 i = 2  bit_reverse(i, 3) = 2
 i = 3  bit_reverse(i, 3) = 6
 i = 4  bit_reverse(i, 3) = 1
 i = 5  bit_reverse(i, 3) = 5
 i = 6  bit_reverse(i, 3) = 3
 i = 7  bit_reverse(i, 3) = 7
gustav@blanc:../src 20:38:46 !602 $
```

A clever compiler should be able to recognise that the whole operation here can be performed in situ, i.e., on `temp`. If converted directly into appropriate hexadecimal instructions, it can be done very quickly.

We're now ready to begin discussion of the Danielson-Lanczos code.

The code comprises two parts. The first one rearranges the data using the reverse-bit ordering. The second part of the code builds up the multi-point Fourier Transform starting from single-point transforms.

Here is the first part of the code, which is going to do Fast Fourier Transform in-situ:

```
  SUBROUTINE fft(x)
    IMPLICIT NONE

    COMPLEX(kind=8), DIMENSION(0:), INTENT(inout) :: x

    INTEGER :: length, number_of_bits, i, j

    length = SIZE(x)
    number_of_bits = INT(LOG(REAL(length))/LOG(2.0))
    IF (2**number_of_bits .NE. length) THEN
       WRITE(*,*) 'fft: error: input data length must be a power of 2'
       STOP 10
    ELSE
       DO i = 1, length-2
          j = bit_reverse(i, number_of_bits)
          IF (j .GT. i) CALL swap(x(i), x(j))
       END DO
    END IF
    ...
```

Observe a particular trick I have exploited here: regardless of how the input data is indexed in the calling program, inside `subroutine fft`, the array is always numbered from `0` through `length - 1`, where `length` is the length of the array. This is accomplished by declaring `x` to be `DIMENSION(0:)`. Also observe that I do not bother to `bit_reverse` `i=0` and `i=length-1`, because these are guaranteed to be two obvious palindromes: `00...0` and `11...1`.

The Danielson-Lanczos part looks as follows, and both at first as well as at consecutive looks it is quite hard to understand:

```
mmax = 1
DO
   IF ( length .LE. mmax) EXIT
   istep = 2 * mmax
   theta = - pi/(mmax)
   wp=CMPLX(-2.0_8 * SIN(0.5_8*theta)**2, SIN(theta), kind=8)
   w=CMPLX(1.0_8, 0.0_8, kind=8)
   DO m = 1, mmax
      ws=w
      DO i=m-1, length-1, istep
         j=i+mmax
         temp = ws*x(j)
         x(j) = x(i) - temp
         x(i) = x(i) + temp
      END DO
      w = w*wp + w
   END DO
   mmax = istep
END DO
```

The easiest way to figure out how this works, is to instrument the code and make it print on standard output what it is doing. So I've done just that and here is what I got:

```
outer do loop: mmax = 1 istep = 2
   middle do loop: m = 1 ws = (1.,0.E+0)
      inner do loop: i = 1 j = 2
      inner do loop: i = 3 j = 4
      inner do loop: i = 5 j = 6
      inner do loop: i = 7 j = 8
outer do loop: mmax = 2 istep = 4
   middle do loop: m = 1 ws = (1.,0.E+0)
      inner do loop: i = 1 j = 3
      inner do loop: i = 5 j = 7
   middle do loop: m = 2 ws = (2.22044604925031308E-16,-1.)
      inner do loop: i = 2 j = 4
      inner do loop: i = 6 j = 8
outer do loop: mmax = 4 istep = 8
   middle do loop: m = 1 ws = (1.,0.E+0)
      inner do loop: i = 1 j = 5
   middle do loop: m = 2 ws = (0.70710678118654746,-0.70710678118654746)
      inner do loop: i = 2 j = 6
   middle do loop: m = 3 ws = (0.E+0,-0.99999999999999978)
      inner do loop: i = 3 j = 7
   middle do loop: m = 4 ws = (-0.70710678118654735,-0.70710678118654735)
      inner do loop: i = 4 j = 8
```

In this print-out I have shifted i and j up by 1 so that we can compare what comes out with equations (2.16), (2.17), and (2.18).

So the first thing we notice is that we evaluate 4 2-point transforms, all with the same ω factor, which is 1. This corresponds to ${}^2\boldsymbol{D}_1 = 1$

In the next step we evaluate two 4-point transforms and this time there are two different ω factors, which are 1 and $-i$. Observe that this corresponds to

${}^4\boldsymbol{D}_2$, which is

$$\begin{pmatrix} 1 & 0 \\ 0 & \omega_4 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & e^{-2\pi i/4} \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & -i \end{pmatrix}$$

In the last step we evaluate one 4 point transform, and this time we have 4 different ω factors, which correspond to ${}^8\boldsymbol{D}_4$, which is:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \omega_8 & 0 & 0 \\ 0 & 0 & \omega_8^2 & 0 \\ 0 & 0 & 0 & \omega_8^3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & e^{-\pi i/4} & 0 & 0 \\ 0 & 0 & e^{-\pi i/2} & 0 \\ 0 & 0 & 0 & e^{-\pi i 3/4} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{1}{\sqrt{2}} - \frac{i}{\sqrt{2}} & 0 & 0 \\ 0 & 0 & -i & 0 \\ 0 & 0 & 0 & -\frac{1}{\sqrt{2}} - \frac{i}{\sqrt{2}} \end{pmatrix}$$

The ω factors are calculated here manually by exploiting a trigonometric recurrence on reals and then constructing corresponding floating point numbers. This is a very efficient way of doing thing, but an obscure one too.

The code that implements the Danielson-Lanczos algorithm discussed here is a very old one. It was written years ago by N. Brenner of Lincoln Laboratories.

As you compare it against our recursive code, you must appreciate the clarity of the recursive algorithm.

Here is the full Danielson-Lanczos program:

```
PROGRAM danielson_lanczos

  IMPLICIT NONE

  REAL(kind=8), PARAMETER :: pi = 3.141592653589793_8
  INTEGER :: i
  COMPLEX(kind=8), DIMENSION(8) :: x, y

  x = (/ (i, i=0,7) /); x = 2.0_8 * pi / 8.0_8 * x; x = SIN(x) + COS(2 * x) &
       - SIN(3 * x)
  y = x
  CALL fft(y)
  y = 2.0_8 * pi / 8.0_8 * y
  WRITE(*,'(8 ("(", f6.3, ",", f7.3, ")", /))') x
  WRITE(*,'(8 ("(", f6.3, ",", f7.3, ")", /))') y
  STOP 0

CONTAINS

  FUNCTION bit_reverse(i, size)
    INTEGER :: bit_reverse
    INTEGER, INTENT(in) :: i, size

    INTEGER :: length, temp

    temp = i
    DO length = size, 2, -1
       temp = ISHFTC(temp, -1, length)
    END DO
    bit_reverse = temp

  END FUNCTION bit_reverse
  SUBROUTINE swap (a, b)
    COMPLEX(kind=8) :: a, b
    COMPLEX(kind=8) :: hold

    hold = a
    a = b
    b = hold

  END SUBROUTINE swap

  SUBROUTINE fft(x)

    IMPLICIT NONE

    COMPLEX(kind=8), DIMENSION(0:), INTENT(inout) :: x

    DOUBLE PRECISION, PARAMETER :: pi = 3.141592653589793_8
    COMPLEX(kind=8) :: wp, w, ws, temp
    REAL(kind=8) :: theta
    INTEGER :: length, number_of_bits, i, j, mmax, istep, m

    length = SIZE(x)
    number_of_bits = INT(LOG(REAL(length))/LOG(2.0))
    IF (2**number_of_bits .NE. length) THEN
       WRITE(*,*) 'fft: error: input data length must be a power of 2'
       STOP 10
    ELSE
       DO i = 1, length-2
          j = bit_reverse(i, number_of_bits)
          IF (j .GT. i) CALL swap(x(i), x(j))
       END DO
    END IF

    mmax = 1
    DO
       IF ( length .LE. mmax) EXIT
       istep = 2 * mmax
       theta = - pi/(mmax)
       wp=CMPLX(-2.0_8 * SIN(0.5_8*theta)**2, SIN(theta), kind=8)
       w=CMPLX(1.0_8, 0.0_8, kind=8)
       DO m = 1, mmax
          ws=w
          DO i=m-1, length-1, istep
             j=i+mmax
             temp = ws*x(j)
             x(j) = x(i) - temp
             x(i) = x(i) + temp
          END DO
          w = w*wp + w
       END DO
       mmax = istep
    END DO

  END SUBROUTINE fft

END PROGRAM danielson_lanczos
```

And here is how this code compiles and runs:

```
gustav@blanc:../src 00:10:38 !565 $ f90 -o lanczos lanczos.f90
gustav@blanc:../src 00:10:53 !566 $ ./lanczos
( 1.000,  0.000)
( 0.000,  0.000)
( 1.000,  0.000)
( 0.000,  0.000)
( 1.000,  0.000)
( 0.000,  0.000)
(-3.000,  0.000)
( 0.000,  0.000)

( 0.000,  0.000)
( 0.000, -3.142)
( 3.142,  0.000)
( 0.000,  3.142)
( 0.000,  0.000)
( 0.000, -3.142)
( 3.142,  0.000)
( 0.000,  3.142)

 STOP 0
gustav@blanc:../src 00:10:55 !567 $
```

2.2.3 Fast Fourier Transform in Parallel

Below is the parallel version of our Danielson-Lanczos program, for a 2 dimensional data.

```
PROGRAM danielson_lanczos

  IMPLICIT NONE

  REAL(kind=8), PARAMETER :: pi = 3.141592653589793_8
  INTEGER :: i, j
  COMPLEX(kind=8), DIMENSION(0:7, 0:7) :: x, y
  REAL(kind=8) :: dx, dy

  DO i = 0,7
     DO j = 0,7
        dx = 2.0_8 * pi / 8.0_8 * i
        dy = 2.0_8 * pi / 8.0_8 * j
        x(i,j) = CMPLX(SIN(dx), 0.0_8, kind=8) * CMPLX(SIN(dy), 0.0_8, kind=8)
     END DO
  END DO
  y = x
  CALL fft(y); y = TRANSPOSE(y); CALL fft(y); y = TRANSPOSE(y)
  y = -2.0_8 * pi / 64.0_8 * y
  WRITE(*,'(8 ( 8 ("(", f6.2, ",", f6.2, ") "), /))') x
  WRITE(*,'(8 ( 8 ("(", f6.2, ",", f6.2, ") "), /))') y
  STOP 0
```

```
CONTAINS

  FUNCTION bit_reverse(i, size)

    IMPLICIT NONE

    INTEGER :: bit_reverse
    INTEGER, INTENT(in) :: i, size

    INTEGER :: length, temp

    temp = i
    DO length = size, 2, -1
       temp = ISHFTC(temp, -1, length)
    END DO
    bit_reverse = temp

  END FUNCTION bit_reverse

  SUBROUTINE swap (a, b)

    IMPLICIT NONE

    COMPLEX(kind=8), DIMENSION(:) :: a
    COMPLEX(kind=8), DIMENSION(SIZE(a)) :: b
    COMPLEX(kind=8), DIMENSION(:), ALLOCATABLE :: hold

    ALLOCATE(hold(SIZE(a)))
    hold = a
    a = b
    b = hold
    DEALLOCATE(hold)

  END SUBROUTINE swap

  SUBROUTINE fft(x)

    IMPLICIT NONE

    COMPLEX(kind=8), DIMENSION(0:, 0:), INTENT(inout) :: x

    DOUBLE PRECISION, PARAMETER :: pi = 3.141592653589793_8
    INTEGER :: length, number_of_bits, i, j, mmax, istep, m
    COMPLEX(kind=8) :: wp, w, ws
    REAL(kind=8) :: theta
    COMPLEX(kind=8), DIMENSION(:), ALLOCATABLE :: temp

    length = SIZE(x, 2)
    number_of_bits = INT(LOG(REAL(length))/LOG(2.0))
    IF (2**number_of_bits .NE. length) THEN
       WRITE(*,*) 'fft: error: input data length must be a power of 2'
       STOP 10
    ELSE
       DO i = 1, length-2
          j = bit_reverse(i, number_of_bits)
          IF (j .GT. i) CALL swap(x(:,i), x(:,j))
       END DO
    END IF

    ALLOCATE(temp(SIZE(x, 1)))
    mmax = 1
    DO
       IF ( length .LE. mmax) EXIT
       istep = 2 * mmax
       theta = - pi/(mmax)
       wp=CMPLX(-2.0_8 * SIN(0.5_8*theta)**2, SIN(theta), kind=8)
       w=CMPLX(1.0_8, 0.0_8, kind=8)
       DO m = 1, mmax
          ws=w
          DO i=m-1, length-1, istep
             j=i+mmax
             temp = ws*x(:,j)
             x(:,j) = x(:,i) - temp
             x(:,i) = x(:,i) + temp
          END DO
          w = w*wp + w
       END DO
       mmax = istep
    END DO
    DEALLOCATE(temp)

  END SUBROUTINE fft

END PROGRAM danielson_lanczos
```

This program differs very little from our original one-dimensional program. I will outline those differences by discussing `diffs` between the two source files:

1.
   ```
   6,16c6,10
   <    INTEGER :: i, j
   <    COMPLEX(kind=8), DIMENSION(0:7, 0:7) :: x, y
   <    REAL(kind=8) :: dx, dy
   <
   <    DO i = 0,7
   <       DO j = 0,7
   <          dx = 2.0_8 * pi / 8.0_8 * i
   <          dy = 2.0_8 * pi / 8.0_8 * j
   <          x(i,j) = CMPLX(SIN(dx), 0.0_8, kind=8) * CMPLX(SIN(dy), 0.0_8, kind=8)
   <       END DO
   <    END DO
   ---
   >    INTEGER :: i
   >    COMPLEX(kind=8), DIMENSION(8) :: x, y
   >
   >    x = (/ (i, i=0,7) /); x = 2.0_8 * pi / 8.0_8 * x; x = SIN(x) + COS(2 * x) &
   >         - SIN(3 * x)
   ```

   This is the first difference: this time my arrays `x` and `y` are two dimensional. For clarity, instead of using an implicit `DO` loop, I have used explicit calculation, where `dx` and `dy` measure the distance from the origin, (0,0), and the input data is of the form of $\sin x \sin y$.

2.
```
18,21c12,15
<   CALL fft(y); y = TRANSPOSE(y); CALL fft(y); y = TRANSPOSE(y)
<   y = -2.0_8 * pi / 64.0_8 * y
<   WRITE(*,'(8 ( 8 ("(", f6.2, ",", f6.2, ") "), /))') x
<   WRITE(*,'(8 ( 8 ("(", f6.2, ",", f6.2, ") "), /))') y
---
>   CALL fft(y)
>   y = 2.0_8 * pi / 8.0_8 * y
>   WRITE(*,'(8 ("(", f6.3, ",", f7.3, ")", /))') x
>   WRITE(*,'(8 ("(", f6.3, ",", f7.3, ")", /))') y
```

   Here, instead of just calling `fft` once, we call it twice. This is still the same `fft` as before, but this time it operates on whole columns of matrix `x`, not on scalar entries of vector `x`. Every manipulation that we have discussed before is now performed on all column entries at once. This is where the parallelism comes in. There is no communication between different rows, hence, if we were to implement this program in High Performance Fortran, the matrix should be distributed row-wise, i.e., different rows should live on different processors. This way our column manipulations will not require expensive inter-process communications.

   The 2-dimensional Fourier Transform works as follows: first we need to find Fourier Transform in the x direction, and then we need to find Fourier Transform on the first Fourier Transform in the y direction.

   Because our subroutine `fft` does only one direction and it is always x, in order to evaluate Fourier Transform in the y direction we need to transpose the matrix, thus interchanging the directions, and then call subroutine `fft` again.

   The returned result must be transposed back to normal.

   This is the only place where communication is involved in our program. Matrix transpose is a fairly costly operation.

   Observe that I have also changed the value of the scaling constant. In this case I wanted the value to match what we have used in our heat diffusion program for the initial temperature distribution, $T_0(x, y)$, so that we can make a quick check if the program runs and if it returns correct results.

3.
```
43a35,36
>     COMPLEX(kind=8) :: a, b
>     COMPLEX(kind=8) :: hold
45,51d37
<     IMPLICIT NONE
<
<     COMPLEX(kind=8), DIMENSION(:) :: a
<     COMPLEX(kind=8), DIMENSION(SIZE(a)) :: b
<     COMPLEX(kind=8), DIMENSION(:), ALLOCATABLE :: hold
<
<     ALLOCATE(hold(SIZE(a)))
55d40
<     DEALLOCATE(hold)
```

This is the change from subroutine `swap`. This time we're swapping two columns of a matrix, no longer two scalar numbers. The columns must be of the same length, of course, and the auxialiary variable `hold` must be an array too.

This array must be allocated once we know the size of the arrays to be swapped, and then deallocated when it is no longer needed.

4.
```
63c48
<     COMPLEX(kind=8), DIMENSION(0:, 0:), INTENT(inout) :: x
---
>     COMPLEX(kind=8), DIMENSION(0:), INTENT(inout) :: x
66,67c51
<     INTEGER :: length, number_of_bits, i, j, mmax, istep, m
<     COMPLEX(kind=8) :: wp, w, ws
---
>     COMPLEX(kind=8) :: wp, w, ws, temp
69c53
<     COMPLEX(kind=8), DIMENSION(:), ALLOCATABLE :: temp
---
>     INTEGER :: length, number_of_bits, i, j, mmax, istep, m
```

Here we're already looking at declarations in subroutine `fft`. Observe that `x` is now an array of rank 2, or, as I prefer to put it, a two-dimensional array.

Variable `temp` must be made into an array this time. It is going to hold `ws*x(:,j)`. And it must be made allocatable to, because, we don't know its size a priori.

5.
```
71c55
<     length = SIZE(x, 2)
---
>     length = SIZE(x)
```

The length of the array of columns is not obtained by calling `SIZE` on the second dimension of the matrix.

6.
```
79c63
<           IF (j .GT. i) CALL swap(x(:,i), x(:,j))
---
>           IF (j .GT. i) CALL swap(x(i), x(j))
```

Instead of swapping two scalar entries of an array, we swap two whole columns. But if each row lives on its own separate processor, this operation should take exactly the same time as swapping two scalar entries of a one-dimensional array.

7.
```
83d66
<     ALLOCATE(temp(SIZE(x, 1)))
95,97c78,80
<              temp = ws*x(:,j)
<              x(:,j) = x(:,i) - temp
<              x(:,i) = x(:,i) + temp
```

```
---
>               temp = ws*x(j)
>               x(j) = x(i) - temp
>               x(i) = x(i) + temp
103d85
<       DEALLOCATE(temp)
```

Array `temp` must be able to store the whole column, so the number of entries that we must allocate for it corresponds to the number of rows, which is the first dimension in an array (rember that the ordering is always *rows* followed by *columns*).

And this is it. This is all we had to do, to parallelise our one dimensional Fast Fourier Transform program. If the program is to run on the SP and if it is to be compiled by HPF, we need to add the following HPF directive in main:

```
!HPF$ DISTRIBUTE (BLOCK, *) x, y
```

This directive should be matched by corresponding directives in subroutines of our program.

Does this program work?

Here is how it compiles and runs:

```
gustav@blanc:../src 16:51:02 !575 $ f90 -o lanczos-p lanczos-p.f90
gustav@blanc:../src 16:53:40 !576 $ ./lanczos-p
(  0.00,  0.00) (  0.00,  0.00) (  0.00,  0.00) (  0.00,  0.00) (  0.00,  0.00) (  0.00,  0.00) (  0.00,  0.00) (  0.00,  0.00)
(  0.00,  0.00) (  0.50,  0.00) (  0.71,  0.00) (  0.50,  0.00) (  0.00,  0.00) ( -0.50,  0.00) ( -0.71,  0.00) ( -0.50,  0.00)
(  0.00,  0.00) (  0.71,  0.00) (  1.00,  0.00) (  0.71,  0.00) (  0.00,  0.00) ( -0.71,  0.00) ( -1.00,  0.00) ( -0.71,  0.00)
(  0.00,  0.00) (  0.50,  0.00) (  0.71,  0.00) (  0.50,  0.00) (  0.00,  0.00) ( -0.50,  0.00) ( -0.71,  0.00) ( -0.50,  0.00)
(  0.00,  0.00) (  0.00,  0.00) (  0.00,  0.00) (  0.00,  0.00) (  0.00,  0.00) (  0.00,  0.00) (  0.00,  0.00) (  0.00,  0.00)
(  0.00,  0.00) ( -0.50,  0.00) ( -0.71,  0.00) ( -0.50,  0.00) (  0.00,  0.00) (  0.50,  0.00) (  0.71,  0.00) (  0.50,  0.00)
(  0.00,  0.00) ( -0.71,  0.00) ( -1.00,  0.00) ( -0.71,  0.00) (  0.00,  0.00) (  0.71,  0.00) (  1.00,  0.00) (  0.71,  0.00)
(  0.00,  0.00) ( -0.50,  0.00) ( -0.71,  0.00) ( -0.50,  0.00) (  0.00,  0.00) (  0.50,  0.00) (  0.71,  0.00) (  0.50,  0.00)

(  0.00,  0.00) (  0.00,  0.00) (  0.00,  0.00) (  0.00,  0.00) (  0.00,  0.00) (  0.00,  0.00) (  0.00,  0.00) (  0.00,  0.00)
(  0.00,  0.00) (  1.57,  0.00) (  0.00,  0.00) (  0.00,  0.00) (  0.00,  0.00) (  0.00,  0.00) (  0.00,  0.00) ( -1.57,  0.00)
(  0.00,  0.00) (  0.00,  0.00) (  0.00,  0.00) (  0.00,  0.00) (  0.00,  0.00) (  0.00,  0.00) (  0.00,  0.00) (  0.00,  0.00)
(  0.00,  0.00) (  0.00,  0.00) (  0.00,  0.00) (  0.00,  0.00) (  0.00,  0.00) (  0.00,  0.00) (  0.00,  0.00) (  0.00,  0.00)
(  0.00,  0.00) (  0.00,  0.00) (  0.00,  0.00) (  0.00,  0.00) (  0.00,  0.00) (  0.00,  0.00) (  0.00,  0.00) (  0.00,  0.00)
(  0.00,  0.00) (  0.00,  0.00) (  0.00,  0.00) (  0.00,  0.00) (  0.00,  0.00) (  0.00,  0.00) (  0.00,  0.00) (  0.00,  0.00)
(  0.00,  0.00) (  0.00,  0.00) (  0.00,  0.00) (  0.00,  0.00) (  0.00,  0.00) (  0.00,  0.00) (  0.00,  0.00) (  0.00,  0.00)
(  0.00,  0.00) ( -1.57,  0.00) (  0.00,  0.00) (  0.00,  0.00) (  0.00,  0.00) (  0.00,  0.00) (  0.00,  0.00) (  1.57,  0.00)

 STOP 0
gustav@blanc:../src 16:53:42 !577 $
```

Comparison with the result discussed in P573 for $T_0(x, y) = \sin x \sin y$ shows that we're right on the spot, i.e.,

$$\begin{aligned} a_{1,1} &= \pi/2 \approx 1.57 \\ a_{1,7} &= -\pi/2 \\ a_{7,1} &= -\pi/2 \\ a_{7,7} &= \pi/2 \end{aligned}$$

Although in our example we have worked with a square data matrix, subroutine `fft` should work with rectangular matrices too.

2.2.4 One Dimensional Parallel Fast Fourier Transform

Is there a way to parallelise one-dimensional Fast Fourier Transform?

Yes.

The way is as follows. Think of a one dimensional data vector as being a collapsed two-dimensional matrix, whose indexes (j, J), where $j \in [0, m]$ and $J \in [0, M]$ collapse onto $f(Jm + j)$.

The 2-dimensional Discrete Fourier Transform of the matrix would be:

$$\begin{aligned} F(kM + K) &= \sum_{j,J} e^{-2\pi i(kM+K)(Jm+j)/(Mm)} f(Jm + j) \\ &= \sum_{j} \left(e^{-2\pi ijk/m} \left(e^{-2\pi ijK/(Mm)} \left(\sum_{J} e^{-2\pi iJK/M} f(Jm + j) \right) \right) \right) \end{aligned} \tag{2.25}$$

This yields the following prescription:

1. Reshape the original 1-dimensional array into an $m \times M$ array in Fortran column-major order.
2. Parallel FFT on the second index, i.e., columns.
3. Multiply each component by a phase factor $\exp(-2\pi ijK/(Mm))$.
4. Transpose.
5. Parallel FFT on the second index, i.e., still columns.
6. Reshape the two-dimensional array into one-dimensional output.

There is a considerable amount of communication involved in this algorithm, so you may not save much if your communication fabric is slow.

Remember that even on the fastest communication fabrics, e.g., switched SMPs, inter-processor communication operations are at least an order of magnitude slower than local memory accesses. On slower fabrics, e.g., external switched communication networks, inter-processor communications may be 3 or more orders of magnitude slower than local memory access.

In turn, local memory operations may be still between one and two orders of magnitude slower than arithmetic CPU operations.

Moving data around is by far the costliest part of every computational process.

2.2.5 Exercise

Replace PESSL `fft` with your own parallel `fft` in your Thermal Diffusion Program, which you have submitted as the second assignment in P573.

2.3 Computerised Tomography

This chapter is based on "An Introduction to High-Performance Scientific Computing" by Fosdick, Jessup, Schauble, and Domik, The MIT Press, 1996

2.3.1 The Filtered Backprojection Method

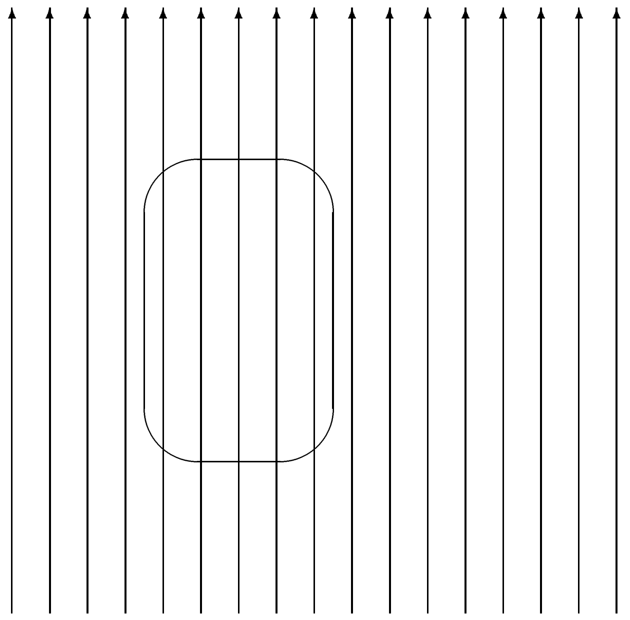

Figure 2.1: X rays penetrating an object in a Computerised Tomography set up.

Assume a computerised tomography set up with rays parallel to the y-axis directed upwards, see Figure 2.1. Assume absorbing density in the plane to be $\mu(x, y)$, where function μ has a compact support. Signal intensity registered on the x axis, $p(\theta = 0, x)$ will then be:

$$p(\theta = 0, x) = \int_{-\infty}^{\infty} \mu(x, y)\, \mathrm{d}y \tag{2.26}$$

Now consider a two-dimensional Fourier transform of function $\mu(x, y)$:

$$M(k, l) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \mu(x, y) e^{-2\pi i(kx+ly)}\, \mathrm{d}x\, \mathrm{d}y \tag{2.27}$$

The $l = 0$ case of this formula is:

$$M(k, 0) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \mu(x, y) e^{-2\pi ikx}\, \mathrm{d}x\, \mathrm{d}y$$

$$= \int_{-\infty}^{\infty} \left(\int_{-\infty}^{\infty} \mu(x,y)\,\mathrm{d}y \right) e^{-2\pi i k x}\,\mathrm{d}x$$
$$= \int_{-\infty}^{\infty} p(\theta = 0, x) e^{-2\pi i k x}\,\mathrm{d}x \tag{2.28}$$

In other words:

> Taking the one-dimensional Fourier Transform of the projection $p(\theta = 0, x)$ along the y axis gives us the $(k, 0)$ line in the Fourier space of $\mu(x, y)$.

We can now rotate our set-up and obtain lines in the Fourier space of $\mu(x, y)$ under any angle by calculating one-dimensional Fourier transforms of measured parallel projections, and in this way effectively reconstructing the whole $M(k, l)$. Once we have $M(k, l)$ we can then obtain $\mu(x, y)$ by taking the *inverse* Fourier Transform of $M(k, l)$.

It is convenient in this case to write it all down in terms of the rotation angle θ.

Consider rotating the system of coordinates as follows:

$$\begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x' \\ y' \end{pmatrix} \tag{2.29}$$

Then:

$$p(\theta, x') = \int_{-\infty}^{\infty} \mu(x', y')\,\mathrm{d}y' \tag{2.30}$$

and

$$\begin{aligned} P(\theta, k') &= \int_{-\infty}^{\infty} p(\theta, x') e^{-2\pi i k' x'}\,\mathrm{d}x' \\ &= \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \mu(x', y') e^{-2\pi i k' x'}\,\mathrm{d}y'\,\mathrm{d}x' \\ &= \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \mu(x', y') e^{-2\pi i k' (x\cos\theta + y\sin\theta)}\,\mathrm{d}x\,\mathrm{d}y = M(\theta, k') \end{aligned} \tag{2.31}$$

The inverse transform of $P(\theta, k')$ yields $\mu(x, y)$:

$$\mu(x, y) = \int_{0}^{2\pi} \int_{0}^{\infty} P(\theta, k') e^{2\pi i k' (x\cos\theta + y\sin\theta)} k'\,\mathrm{d}k'\,\mathrm{d}\theta \tag{2.32}$$

A closer inspection shows that it is not necessary to rotate the apparatus through the whole $[0, 2\pi]$, because swapping the source and the detector does not change the resulting attenuation, i.e.,

$$p(\theta, x') = p(\theta + \pi, L - x') \tag{2.33}$$

where L is the length of the detector. This, in turn, implies that

$$P(\theta, k') = P(\theta + \pi, -k') \tag{2.34}$$

and therefore:

$$\begin{aligned}\mu(x, y) &= \int_0^\pi \int_{-\infty}^{\infty} P(\theta, k') e^{2\pi i k'(x\cos\theta + y\sin\theta)} |k'| \, \mathrm{d}k' \, \mathrm{d}\theta && (2.35)\\ &= \int_0^\pi \int_{-\infty}^{\infty} P(\theta, k') e^{2\pi i k' x'} |k'| \, \mathrm{d}k' \, \mathrm{d}\theta && (2.36)\\ &= \int_0^\pi C(\theta, x') \, \mathrm{d}\theta && (2.37)\end{aligned}$$

where

$$C(\theta, x') = \int_{-\infty}^{\infty} P(\theta, k') e^{2\pi i k' x'} |k'| \, \mathrm{d}k' \tag{2.38}$$

is called a *filtered projection*, and $|k'|$ plays the role of an *inverse filter*: multiplying $P(\theta, k')$ by $|k'|$ increases the influence of $P(\theta, k')$ at high frequencies.

> Integral (2.36) defines the map of density $\mu(x, y)$ in the $x \times y$ plane by *accumulating* all of the filtered projections for all angles θ from 0 to π. Each filtered projection $C(\theta, x')$ contributes to the density along the line of constant $x' = x\cos\theta + y\sin\theta$ for a particular value of θ. And so each filtered projection is *backprojected* into the $x \times y$ plane.

In order to avoid aliasing problems associated with the Nyquist critical frequency $k' = 1/(2\Delta x')$, we shall introduce a new filter function defined as follows:

$$B(k') = \begin{cases} |k'|, & |k'| \le 1/(2\Delta x') \\ 0, & \text{otherwise} \end{cases} \tag{2.39}$$

and convolve it with $P(\theta, k')$ in the expression that defines the filtered projection:

$$C(\theta, x') = \int_{-\infty}^{\infty} P(\theta, k') B(k') e^{2\pi i k' x'} \, \mathrm{d}k' \tag{2.40}$$

The function in the x' space that $B(k')$ corresponds to is $b(x')$:

$$\begin{aligned} b(x') &= \int_{-\infty}^{\infty} B(k') e^{2\pi i k' x'} \, \mathrm{d}k' \\ &= \frac{1}{2(\Delta x')^2} \frac{\sin\left(2\pi x'/(2\Delta x')\right)}{2\pi x'/(2\Delta x')} - \frac{1}{4(\Delta x')^2} \left(\frac{\sin\left(\pi x'/(2\Delta x')\right)}{\pi x'/(2\Delta x')} \right)^2 \end{aligned} \tag{2.41}$$

Now, remember that a Fourier transform of a convolution is equal to a product of Fourier transforms. Since $P(\theta, k')$ and $B(k')$ are Fourier transforms of $p(\theta, x')$

and $b(x')$, their product is equal to Fourier transform of a convolution of the latter two functions:

$$\begin{aligned} P(\theta,k')B(k') &= \int_{-\infty}^{\infty} p(\theta,x')e^{-2\pi i k' x'}\,\mathrm{d}x' \times \int_{-\infty}^{\infty} b(x')e^{-2\pi i k' x'}\,\mathrm{d}x' \\ &= \int_{-\infty}^{\infty} \left(\int_{-\infty}^{\infty} p(\theta,x)b(x'-x)\,\mathrm{d}x \right) e^{-2\pi i k' x'}\,\mathrm{d}x' \end{aligned} \tag{2.42}$$

Since $C(\theta,x')$ is the inverse Fourier Transform of $P(\theta,k')B(k')$, the Fourier Transform in the equation (2.42) gets undone and we're simply left with:

$$C(\theta,x') = \int_{-\infty}^{\infty} p(\theta,x)b(x'-x)\,\mathrm{d}x \tag{2.43}$$

In other words,

> *the* filtered projection $C(\theta,x')$ *is the* convolution *of* $p(\theta,x')$ *and* $b(x')$.

2.3.2 A Discrete Formulation of Filtered Backprojection

Our first step is to replace

$$\mu(x,y) = \int_0^{\pi} C(\theta,x')\,\mathrm{d}\theta$$

with

$$\mu(x,y) \approx \frac{\pi}{N}\sum_{m=0}^{N-1} C\left(\theta_m,x'\right) \tag{2.44}$$

where a single pair (θ_m, x'_n) yields such (x,y) that

$$x' = x\cos\theta + y\sin\theta \tag{2.45}$$

so that going through all measured values of (θ_m, x'_n) will eventually cover the whole plane $x \times y$.

Some comments:

1. Points for small values of k' are much denser than points for large values of k', hence the importance of the filtering function. Filtering $M(\theta,k')$ with $|k'|$ weights the data to compensate for the sparser fill of the frequency domain at larger $|k'|$.

2. Filter $B(k')$ does the same job, whereas its relative $b(x')$ is equivalent, but operates in the spatial domain.

The next step is to discretise

$$C(\theta, x') = \int_{-\infty}^{\infty} p(\theta, x) b(x' - x)\, \mathrm{d}x$$

with

$$C(\theta_m, x'_n) = \sum_{k=-\infty}^{\infty} p(\theta_m, x'_k) b(x'_n - x'_k) \Delta x' \tag{2.46}$$

But the index k has only meaning for x'_0 through x'_{K-1}, and the positions for which we have any measurements of $p(\theta, x')$ are $x'_k = x'_0 + k\Delta x'$.

Convolution is a symmetric operation, i.e., $p \star b = b \star p$. This can be proven simply by subsitution. This means that we can move $x'_n - x'_k$ to p getting:

$$C(\theta_m, x'_n) = \Delta x' \sum_{k=-(K-1)}^{K-1} p(\theta_m, x'_n - x'_k) b(x'_k) \tag{2.47}$$

where we have restricted k to run between $-(K-1)$ and $(K-1)$, because, for example, for $x'_n = x_0$ we can subtract a negative $x'_{-(K-1)}$ and still stay within the measured region, and for $x'_n = x'_{K-1}$ we can subtract x'_{K-1} and go back to x'_0. But we are still going to hit some combinations of x'_n and x'_k that are going to push us beyond the physically meaningful region $[x'_0, x'_{K-1}]$.

What then? We are simply going to *pad* our vector p with zeros.

The discrete values of b_k are going to be:

$$b_k = b(x'_k) = \begin{cases} 1/(2\Delta x')^2, & \text{for } k = 0 \\ 0, & \text{for even } k \\ -1/(k\pi\Delta x')^2 & \text{for odd } k \end{cases} \tag{2.48}$$

In summary:

1. to compute $C(\theta_m, x'_n)$ for a given n we have to evaluate a scalar product of p and b, both of length $2K - 1$
2. to compute $C(\theta_m, x'_n)$ for all n will have to perform $K \times (2K-1) \approx \mathcal{O}(K^2)$ floating point multiplications.

This is going to be expensive.

Because of the low cost of FFT it is cheaper to

1. Take FFT of a padded data function $p = \left(p_{-(K-1)}, \ldots, p_{K-1}\right)$ to form $P(\theta_m, k')$
2. Discretise
$$C(\theta, x') = \int_{-\infty}^{\infty} P(\theta, k') B(k') e^{2\pi i k' x'}\, \mathrm{d}k'$$

This algorithm is going to be cheaper, because its major cost is associated with FFTs and these will run like $(2K-1)\mathcal{O}(2K-1)$.

What next?

1. For a given angle θ_m the vector $C(\theta_m, x'_n)$ gives values of C at points (x, y) such that
$$x_{m,n} \cos\theta_m + y_{m,n} \sin\theta_m = x'_n \tag{2.49}$$

2. The filtered projection values are mapped onto their closest grid box (x_g, y_g), and then summed within that box to produce density $\mu(x_g, y_g)$.

2.4 FFTPACK

FFTPACK is a package of Fortran-77 subroutines for the Fast Fourier Transform of periodic and other symmetric sequences developed by Paul N. Swarztrauber from the National Center for Atmospheric Research in Boulder, Colorado. The project was sponsored by the National Science Foundation.

The subroutines are transferrable by WWW and FTP from the Netlib site:

```
http://www.netlib.org/
```

There are 12 subroutines in the package, plus 7 auxiliary subroutines, which serve to initialize other subroutines, 19 subroutines in total. The essential subroutines are as follows:

rfftf forward transform of a real periodic sequence

rfftb backward transform of a real coefficient array

ezfftf a simplified real periodic forward transform

ezfftb a simplified real periodic backward transform

sint sine transform of a real odd sequence

cost cosine transform of a real even sequence

sinqf forward sine transform with odd wave numbers

sinqb unnormalized inverse of sinqf

cosqf forward cosine transform with odd wave numbers

cosqb unnormalized inverse of cosqf

cfftf forward transform of a complex periodic sequence

cfftb unnormalized inverse of cfftf

See, e.g.,

```
http://beige.ucs.indiana.edu/FFTPACK/doc
```

for more information about those routines.

Chapter 3

Eigensystems

3.1 Introduction

Our example diffusion code, based on Fourier's original paper and on the IBM example code, which we have discussed in P573, called two important subroutines from the PESSL library: the first one was the FFT subroutine, and we have dedicated the previous chapter, 2, to Fast Fourier Transform, the second one was the Eigenvalue problem routine.

At this stage you should review section "The Eigenvalue Problem" in P573 before proceeding.

The Eigenvalue problem, in short, is about finding a non-zero solution to the following equation:

$$\boldsymbol{A} \cdot \boldsymbol{x} = \lambda \boldsymbol{x} \tag{3.1}$$

As I have already mentioned before, if $\boldsymbol{A}$ is a *normal* matrix, i.e., if

$$\boldsymbol{A} \cdot \boldsymbol{A}^\dagger = \boldsymbol{A}^\dagger \cdot \boldsymbol{A} \tag{3.2}$$

then there exists a rotation $\boldsymbol{\Lambda} \in \mathrm{SO}(n)$ such that

$$\boldsymbol{\Lambda}^{-1} \cdot \boldsymbol{A} \cdot \boldsymbol{\Lambda} = \mathrm{diag}\,(\lambda_1, \ldots, \lambda_n) \tag{3.3}$$

If $\boldsymbol{A}$ is *defective*, i.e., if it is *not* normal, then there is still a linear transformation $\boldsymbol{X}$ such that

$$\boldsymbol{X}^{-1} \cdot \boldsymbol{A} \cdot \boldsymbol{X} = \mathrm{diag}\,(\lambda_1, \ldots, \lambda_n) \tag{3.4}$$

but this transformation is no longer of $\mathrm{SO}(n)$.

But whichever is the case the columns of either $\boldsymbol{\Lambda}$ or $\boldsymbol{X}$ are simply eigenvectors of $\boldsymbol{A}$ that correspond to appropriate λs.

There are actually two types of eigenvectors, which are referred to as left and right eigenvectors. If you took to heart what I have told you about vectors and forms in the past, you will recognise now that right eigenvectors are vectors, and left eigenvectors are forms, and that an orthogonality relationship exists between the two:

$$\begin{aligned}\boldsymbol{A}\cdot\boldsymbol{x}_R &= \lambda\boldsymbol{x}_R\\ \boldsymbol{\chi}_L\cdot\boldsymbol{A} &= \lambda\boldsymbol{\chi}_L\\ \langle\boldsymbol{\chi}^i,\boldsymbol{x}_k\rangle &= \delta^i{}_k\end{aligned}$$

This is the case even for the *defective* matrices, although for defective matrices it may happen that neither $\{\boldsymbol{x}_k\}$ nor $\{\boldsymbol{\chi}^i\}$ form an (orthogonal) basis.

The way most eigenproblem routines work is that matrix $\boldsymbol{A}$ is nudged, usually iteratively, towards a diagonal form by a sequence of linear transformations:

$$\begin{aligned}&\boldsymbol{A}\\ &\boldsymbol{P}_1^{-1}\cdot\boldsymbol{A}\cdot\boldsymbol{P}_1\\ &\boldsymbol{P}_2^{-1}\cdot\boldsymbol{P}_1^{-1}\cdot\boldsymbol{A}\cdot\boldsymbol{P}_1\cdot\boldsymbol{P}_2\\ &\ldots\\ &\boldsymbol{P}_k^{-1}\cdot\ldots\cdot\boldsymbol{P}_2^{-1}\cdot\boldsymbol{P}_1^{-1}\cdot\boldsymbol{A}\cdot\boldsymbol{P}_1\cdot\boldsymbol{P}_2\cdot\ldots\cdot\boldsymbol{P}_k\end{aligned}$$

Once the resulting matrix is diagonal, up to the required accuracy, eigenvectors are returned in

$$\boldsymbol{P}_1\cdot\boldsymbol{P}_2\cdot\ldots\cdot\boldsymbol{P}_k$$

The solution of an eigenproblem may get quite complicated and costly. Public domain eigenproblem routines are available within LAPACK, e.g.,:

CGEEV compute for an $N\times N$ complex nonsymmetric matrix $\boldsymbol{A}$ the eigenvalues and, optionally, the left and/or right eigenvectors

CHBEV compute all the eigenvalues and, optionally, eigenvectors of a complex Hermitian band matrix $\boldsymbol{A}$

CPTEQR compute all eigenvalues and, optionally, eigenvectors of a symmetric positive definite tridiagonal matrix

DGEEV compute for an $N\times N$ real nonsymmetric matrix $\boldsymbol{A}$, the eigenvalues, and, optionally, the lef and/or right eigenvectors

DSBEV compute all the eigenvalues and, optionally, eigenvectors of a real symmetric band matrix $\boldsymbol{A}$

DSPEV compute all the eigenvalues and, optionally, eigenvectors of a real symmetric matrix $\boldsymbol{A}$ in packed storage

DSYEV compute all eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix $\boldsymbol{A}$

There are many, many more in there specialised for various combinations of symmetries and requirements.

Normally you would want such specialised routines to cover at least the following:

- real, symmetric, tridiagonal
- real, symmetric, banded
- real, symmetric
- real, nonsymmetric
- complex, Hermitian
- complex, non-Hermitian

The purpose of those specialisations is to save time and storage, an important considerations especially when the matrices grow very large.

3.2 Jacobi Transformations of a Symmetric Matrix

The orthogonal transformations $\boldsymbol{P}_{pq}$ annihilate element (p, q) of an object matrix.

Successive transformations undo previously set zeros, but the off-diagonal terms eventually get smaller and smaller, until they get *nearly* zero, and the only thing that's left is the diagonal with eigenvalues.

Taking the product of all the transformations $\boldsymbol{P}_{pq}$ yields the matrix of eigenvectors.

The method is absolutely foolproof for symmetric real matrices. But it is slow. Painfully slow for matrices larger than 10×10.

It is a simple and stable algorithm though, and it parallelises well too.

The Jacobi rotation matrix has the form:

$$\boldsymbol{P}_{pq} = \begin{pmatrix} 1 & & & & & & \\ & \ldots & & & & & \\ & & c & \cdots & s & & \\ & & \vdots & 1 & \vdots & & \\ & & -s & \cdots & c & & \\ & & & & & \ldots & \\ & & & & & & 1 \end{pmatrix} \tag{3.5}$$

All diagonal elements are 1 with the exception of (p, p) and (q, q), which are c. The (p, q) element is s and the (q, p) element is $-s$. All other elements are 0. Furthermore:

$$c = \cos\varphi \tag{3.6}$$

$$s = \sin\varphi \tag{3.7}$$

hence

$$c^2 + s^2 = 1 \tag{3.8}$$

Because $\boldsymbol{P}_{pq} \in \mathrm{SO}(n)$, $\boldsymbol{P}_{pq}^{-1} = \boldsymbol{P}_{pq}^{T}$ and so

$$\boldsymbol{A}' = \boldsymbol{P}_{pq}^{T} \cdot \boldsymbol{A} \cdot \boldsymbol{P}_{pq} \tag{3.9}$$

This operation will affect only rows p and q and columns p and q leaving the rest of the matrix unchanged.

It is quite easy to see what the effect of equation (3.9) is going to be on selected terms.

First we need to come up with an expression that describes a generic term of matrix $\boldsymbol{P}_{pq}$ in terms of Kronecker deltas:

$$\begin{aligned} P_{ij} &= \delta_{ij} + \delta_{ip}\delta_{jp}(c-1) + \delta_{iq}\delta_{jq}(c-1) + \delta_{ip}\delta_{jq}s - \delta_{jp}\delta_{iq}s \\ &= \delta_{ij} + \left(\delta_{ip}\delta_{jp} + \delta_{iq}\delta_{jq}\right)(c-1) + \left(\delta_{ip}\delta_{jq} - \delta_{iq}\delta_{jp}\right)s \end{aligned} \tag{3.10}$$

Now we can evaluate a'_{rp} for, say, $r \neq p$ and $r \neq q$ (assume summation over dummy indexes i and j)

$$\begin{aligned} a'_{rp} &= P^{-1}_{ri} a_{ij} P_{jp} = P_{ir} a_{ij} P_{jp} \\ &= P_{ir} a_{ij} \left(\delta_{jp} + \left(\delta_{jp}\delta_{pp} + \delta_{jq}\delta_{pq}\right)(c-1) + \left(\delta_{jp}\delta_{pq} - \delta_{jq}\delta_{pp}\right)s\right) \\ &= P_{ir} a_{ij} \left(\delta_{jp} + \delta_{jp}(c-1) - \delta_{jq}\delta_{pp}s\right) \\ &= P_{ir}\left(a_{ip} + (c-1)a_{ip} - s a_{iq}\right) = P_{ir}\left(c a_{ip} - s a_{iq}\right) \\ &= \left(\delta_{ir} + \left(\delta_{ip}\delta_{rp} + \delta_{iq}\delta_{rq}\right)(c-1) + \left(\delta_{ip}\delta_{rq} - \delta_{iq}\delta_{rp}\right)s\right)\left(c a_{ip} - s a_{iq}\right) \\ &= \delta_{ir}\left(c a_{ip} - s a_{iq}\right) = \left(c a_{rp} - s a_{rq}\right) \end{aligned} \tag{3.11}$$

In summary:

$$\begin{aligned} a'_{rp} &= c a_{rp} - s a_{rq} \quad \text{for} \quad r \neq p, r \neq q && (3.12) \\ a'_{rq} &= c a_{rq} + s a_{rp} \quad \text{for} \quad r \neq p, r \neq q && (3.13) \\ a'_{pp} &= c^2 a_{pp} + s^2 a_{qq} - 2sc a_{pq} && (3.14) \\ a'_{qq} &= s^2 a_{pp} + c^2 a_{qq} + 2sc a_{pq} && (3.15) \\ a'_{pq} &= \left(c^2 - s^2\right) a_{pq} + sc\left(a_{pp} - a_{qq}\right) = a'_{qp} && (3.16) \end{aligned}$$

The purpose of the Jacobi rotation $\boldsymbol{P}_{pq}$ is to kill a'_{pq}. Thus:

$$a'_{pq} = 0 = \left(c^2 - s^2\right) a_{pq} + sc\left(a_{pp} - a_{qq}\right) \tag{3.17}$$

which implies

$$\frac{c^2 - s^2}{sc} = \frac{a_{qq} - a_{pp}}{a_{pq}} \tag{3.18}$$

Now, observe that:

$$\begin{aligned} \cos 2\varphi &= \cos^2\varphi - \sin^2\varphi \\ \sin 2\varphi &= 2\sin\varphi\cos\varphi \end{aligned}$$

This means that if we divide both sides of equation (3.18) by 2, we'll get:

$$\cot 2\varphi = \frac{\cos 2\varphi}{\sin 2\varphi} = \frac{c^2 - s^2}{2sc} = \frac{a_{qq} - a_{pp}}{2a_{pq}} \doteq \theta \tag{3.19}$$

Let us call this $\cot 2\varphi$ a θ for convenience.

Next recall the following simple algebraic identity:

$$\tan^2 \varphi + 2 \tan \varphi \cot 2\varphi - 1 = 0 \tag{3.20}$$

This is easy to see. Since $\tan \varphi = s/c$, we have:

$$\tan^2 \varphi + 2 \tan \varphi \cot 2\varphi - 1 = \frac{s^2}{c^2} + 2\frac{s}{c}\frac{c^2 - s^2}{2sc} - 1 = \frac{s^2 + c^2 - s^2}{c^2} - 1 = 1 - 1 = 0 \tag{3.21}$$

Now, denote $\tan \varphi$ by t, so that the equation looks as follows:

$$t^2 + 2\theta t - 1 = 0 \tag{3.22}$$

Solving equation (3.22) with respect to t yields the following:

$$\begin{aligned}\Delta &= 4\theta^2 - 4 \cdot 1 \cdot (-1) = 4\theta^2 + 4 \\ t_\pm &= \frac{-2\theta \pm 2\sqrt{\theta^2 + 1}}{2} = -\theta \pm \sqrt{\theta^2 + 1}\end{aligned}$$

Now, this solution can be rewritten in a computationally more convenient form. Consider the + case:

$$\begin{aligned}t_+ &= -\theta + \sqrt{\theta^2 + 1} \\ &= -1\frac{\left(\theta - \sqrt{\theta^2 + 1}\right)\left(\theta + \sqrt{\theta^2 + 1}\right)}{\left(\theta + \sqrt{\theta^2 + 1}\right)} \\ &= -1\frac{\theta^2 - \theta^2 - 1}{\theta + \sqrt{\theta^2 + 1}} \\ &= \frac{1}{\theta + \sqrt{\theta^2 + 1}}\end{aligned}$$

The $-$ case yields:

$$\begin{aligned}t_- &= -\theta - \sqrt{\theta^2 + 1} \\ &= -1\frac{\left(\theta + \sqrt{\theta^2 + 1}\right)\left(\theta - \sqrt{\theta^2 + 1}\right)}{\left(\theta - \sqrt{\theta^2 + 1}\right)} \\ &= -1\frac{\theta^2 - \theta^2 - 1}{\theta - \sqrt{\theta^2 + 1}} \\ &= \frac{-1}{-\theta + \sqrt{\theta^2 + 1}}\end{aligned}$$

If we get a positive θ in the $+$ case and a negative θ in the $-$ case we'll end up with the same smaller t that corresponds to an angle $\varphi < \pi/4$. This will yield the most stable reduction. So we can rewrite the formula for t as follows:

$$t = \frac{\text{sign}(\theta)}{|\theta| + \sqrt{\theta^2 + 1}} \tag{3.23}$$

Once we have the t, we get c and s as follows[1]

$$\begin{aligned} c &= \frac{1}{\sqrt{t^2+1}} && (3.24) \\ s &= ct && (3.25) \end{aligned}$$

Equations (3.12) through (3.16) are now rewritten to minimize the round off error and to make them look like the new quantity is equal to the old one plus a small correction. And so,

$$a'_{pq} = a'_{qp} = 0 = (c^2 - s^2)a_{pq} + sc(a_{pp} - a_{qq}) \tag{3.26}$$

by definition. Hence:

$$a_{qq} = \frac{c^2 - s^2}{sc} a_{pq} + a_{pp} \tag{3.27}$$

Then we have

$$\begin{aligned} a'_{pp} &= c^2 a_{pp} + s^2 a_{qq} - 2sca_{pq} \\ &= c^2 a_{pp} + s^2 \left(\frac{c^2 - s^2}{sc} a_{pq} + a_{pp} \right) - 2sca_{pq} \\ &= (c^2 + s^2) a_{pp} + a_{pq} \left(s^2 \frac{c^2 - s^2}{sc} - 2sc \right) \\ &= a_{pp} + a_{pq} \frac{s^2c^2 - s^4 - 2s^2c^2}{sc} \\ &= a_{pp} - ta_{pq} && (3.28) \end{aligned}$$

Similarly from the same equation (3.26) we get:

$$a_{pp} = a_{qq} - \frac{c^2 - s^2}{sc} a_{pq} \tag{3.29}$$

and then

$$\begin{aligned} a'_{qq} &= s^2 a_{pp} + c^2 a_{qq} + 2sca_{pq} \\ &= s^2 \left(a_{qq} - \frac{c^2 - s^2}{sc} a_{pq} \right) + c^2 a_{qq} + 2sca_{pq} \end{aligned}$$

[1]There is an error in the formula for c in the "Numerical Recipes". But the code is fine. You can easily spot the error if you compare the formula with the code actually – assuming that you don't quite remember what it should be.

$$\begin{aligned}
&= (s^2+c^2)a_{qq} + a_{pq}\left(2sc - s^2\frac{c^2-s^2}{sc}\right) \\
&= a_{qq} + a_{pq}\frac{2s^2c^2 - s^2c^2 + s^4}{sc} \\
&= a_{qq} + ta_{pq} \qquad (3.30)
\end{aligned}$$

For a'_{rp} and a'_{rq} the computation is more trivial:

$$\begin{aligned}
a'_{rp} &= ca_{rp} - sa_{rq} \\
&= a_{rp} + (c-1)a_{rp} - sa_{rq} \\
&= a_{rp} - s\left(a_{rq} + \frac{1-c}{s}a_{rp}\right) \qquad (3.31)
\end{aligned}$$

and

$$\begin{aligned}
a'_{rq} &= ca_{rq} + sa_{rp} \\
&= a_{rq} + (c-1)a_{rq} + sa_{rp} \\
&= a_{rq} + s\left(a_{rp} - \frac{1-c}{s}a_{rq}\right) \qquad (3.32)
\end{aligned}$$

where

$$\frac{1-c}{s} = \frac{(1-c)(1+c)}{s(1+c)} = \frac{1-c^2}{s(1+c)} = \frac{s^2}{s(1+c)} = \frac{s}{1+c} = \tau = \tan\frac{\varphi}{2} \qquad (3.33)$$

3.2.1 Convergence of the Jacobi Method

Consider

$$S = \sum_{r\neq s} |a_{rs}|^2 \qquad (3.34)$$

The transformation $S \to S'$ is orthogonal and it results in increase in the sum of the squares of the diagonal elements by $2t^2|a_{pq}|^2$. Consequently, the sum of the squares of the off-diagonal terms must decrease by $2t^2|a_{pq}|^2$. Because S is limited from below, it must be always greater than or equal to zero, and it diminishes with every step, eventually we must converge.

3.2.2 The Eigenvectors in the Jacobi Method

Eventually we end up with a matrix $\boldsymbol{D}$, which is diagonal to requested precision. The obtained transformation is:

$$\boldsymbol{D} = \boldsymbol{V}^T \cdot \boldsymbol{A} \cdot \boldsymbol{V} \qquad (3.35)$$

where

$$\boldsymbol{V} = \boldsymbol{P}_1 \cdot \boldsymbol{P}_2 \cdot \boldsymbol{P}_3 \cdot \ldots \cdot \boldsymbol{P}_n \qquad (3.36)$$

Given matrix $\boldsymbol{V}$ we can compute the new matrix $\boldsymbol{V}'$ as follows:

$$\begin{aligned} v'_{rs} &= v_{rs} \quad \text{for} \quad s \neq p, s \neq q & (3.37)\\ v'_{rp} &= cv_{rp} - sv_{rq} = v_{rp} - s\left(v_{rq} + \tau v_{rp}\right), & (3.38)\\ v'_{rq} &= cv_{rq} + sv_{rp} = v_{rq} + s\left(v_{rp} - \tau v_{rq}\right) & (3.39) \end{aligned}$$

3.2.3 The Strategy in the Jacobi Method

In the original algorithm of 1846 Jacobi would search the whole upper triangle of matrix $\boldsymbol{A}'$ and set the *largest* element to zero. But computers are incapable of finding the largest element simply by glancing at the whole matrix, the way we do. They can only do things like that by looking at each element in separation from others.

Computers are woefully shortsighted.

In our case we'll annihilate elements in a strict order by proceeding down the rows: $\boldsymbol{P}_{12}$, $\boldsymbol{P}_{13}$, $\boldsymbol{P}_{14}$ and so on, then $\boldsymbol{P}_{23}$, $\boldsymbol{P}_{24}$, etc. One such set of $n(n-1)/2$ Jacobi rotations is called a *sweep*.

3.2.4 About Jacobi

Karl Gustav Jacob Jacobi was born on the 10th of December 1804 in Potsdam, and died on the 18th of February, 1851, in Berlin. He was a German mathematician who, with Niels Henrik Abel of Norway, founded the theory of elliptic functions.

In 1827 Jacobi became extraordinary professor and in 1829 ordinary professor of mathematics at the University of Königsberg. He first became known through his work on number theory, which gained the admiration of Carl Gauss, the greatest mathematician of his (and perhaps any) day. Unaware of similar endeavours by Abel, Jacobi formulated a theory of elliptic functions based on four theta functions. The quotients of the theta functions yield the three Jacobian elliptic functions: $s_n(z)$, $c_n(z)$, and $d_n(z)$. His results in elliptic functions were published in Fundamenta Nova Theoriae Functionum Ellipticarum (1829; "New Foundations of the Theory of Elliptic Functions"). In 1832 he demonstrated that just as elliptic functions can be obtained by inverting elliptic integrals, hyperelliptic functions can be obtained by inverting hyperelliptic integrals. This success led him to the formation of the theory of Abelian functions of p variables (where $p \geq 2$).

Jacobi's De Formatione et Proprietatibus Determinantium (1841; "Concerning the Structure and Properties of Determinants") made pioneering contributions to the theory of determinants. He invented the functional determinant (formed from the n^2 differential coefficients of n given functions with n independent variables) that bears his name and has played an important part in many analytical investigations.

Jacobi carried out important research in partial differential equations of the first order and applied them to the differential equations of dynamics. His Vorlesungen über Dynamik (1866; "Lectures on Dynamics") relates his work with differential equations and dynamics. The Hamilton-Jacobi equation plays a significant role in the presentation of quantum mechanics. He also made important studies of Abelian transcendents and the applications of elliptic functions to the theory of numbers.

3.2.5 Example Code

```
SUBROUTINE jacobi(a, d, v, nrot)

  USE nrtype; USE nrutil, ONLY : assert_eq, get_diag, nrerror, unit_matrix, &
       upper_triangle

  IMPLICIT NONE
  INTEGER(i4b), INTENT(out) :: nrot
  REAL(sp), DIMENSION(:), INTENT(out) :: d
  REAL(sp), DIMENSION(:,:), INTENT(inout) :: a
  REAL(sp), DIMENSION(:,:), INTENT(out) :: v

  INTEGER(i4b) :: i, ip, iq, n
  REAL(sp) :: c, g, h, s, sm, t, tau, theta, tresh
  REAL(sp), DIMENSION(SIZE(d)) :: b, z

  n = assert_eq((/SIZE(a, 1), SIZE(a, 2), SIZE(d), SIZE(v, 1), SIZE(v, 2)/), &
       'jacobi')
  CALL unit_matrix(v(:,:))
  b(:) = get_diag(a(:,:))
  d(:) = b(:)
  z(:) = 0.0
  nrot = 0
  DO i = 1, 50
     sm = SUM(ABS(a), mask=upper_triangle(n,n))
     IF(sm == 0.0) RETURN
     tresh = MERGE(0.2_sp*sm/n**2, 0.0_sp, i < 4)
     DO ip=1, n-1
        DO iq=ip+1, n
           g=100.0_sp*ABS(a(ip,iq))
           IF((i > 4) .AND. (ABS(d(ip))+g == ABS(d(ip))) &
                .AND. (ABS(d(iq))+g == ABS(d(iq)))) THEN
              a(ip,iq)=0.0
           ELSE IF (ABS(a(ip,iq)) > tresh) THEN
              h = d(iq)-d(ip)
              IF(ABS(h)+g == ABS(h)) THEN
                 t = a(ip,iq)/h
              ELSE
                 theta = 0.5_sp*h/a(ip,iq)
                 t = 1.0_sp/(ABS(theta)+SQRT(1.0_sp+theta**2))
                 IF(theta < 0.0) t = -t
              END IF
              c = 1.0_sp / SQRT(1+t**2)
              s = t*c
              tau = s/(1.0_sp + c)
              h = t*a(ip, iq)
              z(ip) = z(ip) - h
              z(iq) = z(iq) + h
              d(ip) = d(ip) - h
              d(iq) = d(iq) + h
```

```
            a(ip, iq) = 0.0
            CALL jrotate(a(1:ip-1, ip), a(1:ip-1, iq))
            CALL jrotate(a(ip, ip+1:iq-1), a(ip+1:iq-1, iq))
            CALL jrotate(a(ip, iq+1:n), a(iq, iq+1:n))
            CALL jrotate(v(:,ip), v(:,iq))
            nrot=nrot+1
         END IF
      END DO
   END DO
   b(:) = b(:) + z(:)
   d(:) = b(:)
   z(:) = 0.0
 END DO
 CALL nrerror('too many iterations in jacobi')

CONTAINS

 SUBROUTINE jrotate(a1, a2)
   REAL(sp), DIMENSION(:), INTENT(inout) :: a1, a2
   REAL(sp), DIMENSION(SIZE(a1)) : wk1
   wk1(:) = a1(:)
   a1(:) = a1(:) - s*(a2(:) + a1(:) * tau)
   a2(:) = a2(:) + s*(wk1(:) - a2(:) * tau)
 END SUBROUTINE jrotate

END SUBROUTINE jacobi
```

Let me explain this code now in some detail.

The subroutine computes all eigenvalues and eigenvectors of an $N\times N$ matrix a. The elements of a above the diagonal are destroyed in the process. The eigenvalues are returned on a vector of length N: d, whereas the eigenvectors are returned on v, which a matrix $N \times N$, like a. The total number of Jacobi rotations is returned on nrot.

The subroutine begins with a call to function assert_eq, which is a still to be defined function, whose purpose is to check that a is indeed a square matrix and that the sizes of d and v match a. If these conditions are not met, the function exits with error message that contains the word 'jacobi', otherwise the common dimension N is returned and captured on n:

```
 n = assert_eq((/SIZE(a, 1), SIZE(a, 2), SIZE(d), SIZE(v, 1), SIZE(v, 2)/), &
      'jacobi')
```

Then we initialize v to a unit matrix, extract the diagonal from a and copy it both on an auxiliary array b and on d, which will ultimately return the eigenvalues. Another auxiliary array, z is set to zero. The subroutine could be written without b and z – their use obscures the flow of computation a little, and, I think that it is here only for the sake of improving the numerical accuracy of the computational process, which contains a lot of additions and subtractions:

```
 CALL unit_matrix(v(:,:))
 b(:) = get_diag(a(:,:))
 d(:) = b(:)
 z(:) = 0.0
```

Now we initialize our Jacobi rotation counter to zero and commence Jacobi *sweeps*. The procedure should normally converge in no more than about 5 sweeps. If it takes more, something must be really wrong. This main loop looks as follows:

```
nrot = 0
DO i = 1, 50
   sm = SUM(ABS(a), mask=upper_triangle(n,n))
   IF(sm == 0.0) RETURN
   tresh = MERGE(0.2_sp*sm/n**2, 0.0_sp, i < 4)
   DO ip=1, n-1
      DO iq=ip+1, n
         blah... blah... blah...
      END DO
   END DO
   b(:) = b(:) + z(:)
   d(:) = b(:)
   z(:) = 0.0
END DO
CALL nrerror('too many iterations in jacobi')
```

At the very beginning we sum absolute values of all terms in the upper triangle of matrix $\boldsymbol{A}$ (or a) and if that sum ends up being 0 up to machine accuracy, we return:

$$\mathtt{sm} = \sum_{p=1}^{n-1} \sum_{q=p+1}^{n} |A_{pq}|$$

No special cleaning is necessary at this stage, because all expected results are already in place.

The parameter `tresh` is calculated by calling a Fortran intrinsic `MERGE`, which returns:

- $\frac{1}{5}\frac{1}{N^2}\sum_{p=1}^{n-1}\sum_{q=p+1}^{n} |A_{pq}|$ for the first three sweeps, and
- 0 for the remaining sweeps.

The purpose of this parameter is to annihilate only those terms in the upper triangular of $\boldsymbol{A}$ that stick out in the first three sweeps, and only then commence work on the whole upper triangular.

We skip the discussion of what's inside the double loop that sweeps through the upper triangular of $\boldsymbol{A}$ and go right to what happens after the sweep is done.

What you see there is what those two auxiliary vectors `b` and `z` are for. Vector `z` accumulates ta_{pq} with an appropriate sign ($-$ for pp and $+$ for qq) for every diagonal term that is stored on `b`. All those accumulated contributions are now added to the diagonal, and the new updated diagonal is transferred to `d`. Then vector `z` is cleared and made ready for the new sweep.

You will see that inside the sweep loop the diagonal, which is stored on `d` is updated all the time too, in a similar way to `z`, so, in principle, `d` should contain the same numbers as `b(:) + z(:)`, why then do `d(:) = b(:)`? It is here, I think, that the designers of the code tried to minimize rounding errors that would accumulate on `d(:)` in the process.

If for some reason we've done 50 iterations and the process still hasn't converged, we exit the routine with an error message.

Now let us have a look at what's inside the double loop that sweeps through the upper triangular of $\boldsymbol{A}$.

We begin by testing how large is the element A_{pq}, which is to be annihilated. So first we simply take $g = 100|A_{pq}|$, and then we execute this elaborate IF statement:

```
g=100.0_sp*ABS(a(ip,iq))
IF((i > 4) .AND. (ABS(d(ip))+g == ABS(d(ip))) &
     .AND. (ABS(d(iq))+g == ABS(d(iq)))) THEN
   a(ip,iq)=0.0
ELSE IF (ABS(a(ip,iq)) > tresh) THEN
   blah... blah... blah...
END IF
```

The first clause of IF test for the following condition:

- the sweep number is greater than 4, *and*
- $A_{pp} + 100|A_{pq}| = A_{pp}$, *and*
- $A_{qq} + 100|A_{pq}| = A_{qq}$.

How can $A_{qq} + 100|A_{pq}|$ be equal to A_{qq}? It can be, if $100|A_{pq}|$ is so much smaller than A_{qq} that the value of A_{qq} does not change to within the machine accuracy. In other words what we're saying here is that if the element to be annihilated is so small that the Jacobi rotation will not change the diagonal elements, then don't bother with the rotation at all: just make that off-diagonal element zero.

Otherwise, i.e., if the off-diagonal element needs to be rotated away, then check if the element A_{pq} is greater than the threashold we have evaluated earlier.

That threashold was different from zero only for the first three sweeps, and its value was a kind of an average value for the upper triangular: $\frac{1}{5}\frac{1}{N^2}\sum_{p=1}^{n-1}\sum_{q=p+1}^{n}|A_{pq}|$. So it is here that we implement Jacobi rotation for the off-diagonal elements that stand out during the first three sweeps.

By now we've done enough checking, and enough avoiding, and finally we have to get down to work and perform the actual rotation.

The first step is to evaluate

$$t = \tan\varphi = \frac{\text{sign}(\theta)}{|\theta| + \sqrt{\theta^2+1}} \approx \frac{1}{2\theta}$$

where the approximation holds for very large θ, and where

$$\theta = \cot 2\varphi = \frac{c^2 - s^2}{2sc} = \frac{a_{qq} - a_{pp}}{2a_{pq}}$$

Remember that the diagonal terms a_{qq} and a_{pp} live in d:

```
h = d(iq)-d(ip)
IF(ABS(h)+g == ABS(h)) THEN
   t = a(ip,iq)/h
```

```
ELSE
   theta = 0.5_sp*h/a(ip,iq)
   t = 1.0_sp/(ABS(theta)+SQRT(1.0_sp+theta**2))
   IF(theta < 0.0) t = -t
END IF
```

Now, once we have t, we evaluate c, s, and τ:

$$\begin{aligned} c &= \cos\varphi = \frac{1}{\sqrt{t^2+1}} \\ s &= \sin\varphi = \tan\varphi\cos\varphi = tc \\ \tau &= \tan\frac{\varphi}{2} = \frac{\sin\varphi}{1+\cos\varphi} = \frac{s}{1+c} \end{aligned}$$

```
c = 1.0_sp / SQRT(1+t**2)
s = t*c
tau = s/(1.0_sp + c)
```

The next step is to rotate the diagonal elements, because they are easy to do:

$$\begin{aligned} a'_{pp} &= a_{pp} - ta_{pq} \\ a'_{qq} &= a_{qq} + ta_{pq} \end{aligned}$$

```
h = t*a(ip, iq)
z(ip) = z(ip) - h
z(iq) = z(iq) + h
d(ip) = d(ip) - h
d(iq) = d(iq) + h
```

Observe that it is here that we collect the total change to a_{ii} per sweep on `z`, whereas the diagonal terms themselves are updated for every sweep on `d`.

At this stage we won't need a_{pq} any more, because it is no longer used explicitly in rotating a_{rp} and a_{rq}, so we annihilate it:

```
a(ip, iq) = 0.0
```

Finally we rotate a_{rp} and a_{rq}:

$$\begin{aligned} a'_{rp} &= a_{rp} - s(a_{rq} + \tau a_{rp}) \\ a'_{rq} &= a_{rq} + s(a_{rp} - \tau a_{rq}) \end{aligned}$$

by calling an auxiliary subroutine `jrotate`:

```
CALL jrotate(a(1:ip-1, ip), a(1:ip-1, iq))
CALL jrotate(a(ip, ip+1:iq-1), a(ip+1:iq-1, iq))
CALL jrotate(a(ip, iq+1:n), a(iq, iq+1:n))
```

These three call correspond to three subsets of $\boldsymbol{A}$ that need to be rotated separately, so that we don't hit the point (p, q) itself.

The first call rotates the two vertical lines: one that stretches from (p, q) upwards, and the other one, parallel to it and slightly to the left, that stretches from the diagonal upwards. Both are of equal length.

The second call rotates the two perpendicular lines between the point (p, q) and the diagonal: one is horizontal and the other is vertical. Again, they are of equal length.

Finall, the third call rotates the two horizontal lines: one that stretches from (p, q) to the right boundary, and the other one slightly below, that stretches from the right boundary to the diagonal. Both are of equal length.

Having rotated a_{rp} and a_{rq} we rotate the matrix of eigenvectors `v`. Remember that they are rotated in a much the same way as a_{rp} and a_{rq}, i.e.,

$$\begin{aligned} v'_{rp} &= v_{rp} - s(v_{rq} + \tau v_{rp}) \\ v'_{rq} &= v_{rq} + s(v_{rp} - \tau v_{rq}) \end{aligned}$$

This is implemented simply by calling:

```
CALL jrotate(v(:,ip), v(:,iq))
```

After we have rotated matrices $\boldsymbol{A}$ and $\boldsymbol{V}$, we increment the Jacobi rotation counter `nrot`:

```
nrot=nrot+1
```

The auxiliary subroutine `jrotate` implements the equations:

$$\begin{aligned} a'_{rp} &= a_{rp} - s(a_{rq} + \tau a_{rp}) \\ a'_{rq} &= a_{rq} + s(a_{rp} - \tau a_{rq}) \end{aligned}$$

Observe one small complication. Because the old a_{rp} is needed by the second equation we must save it before we alter it with the first equation. Subroutine `jrotate` performs the computation in parallel as follows:

```
wk1(:) = a1(:)
a1(:) = a1(:) - s*(a2(:) + a1(:) * tau)
a2(:) = a2(:) + s*(wk1(:) - a2(:) * tau)
```

Observe that the parallelism is both in the horizontal and in the vertical direction of matrix $\boldsymbol{A}$. There is a fair amount of communication involved too. This algorithm will run well on a vector processor or on an SMP, but not so well on an MPP.

In any case, the code presented in this section shows all parallel operations explicitly and clearly. It is now up to the compiler to make use of it.

Unfortunately, there are no machines on the market at present that can operate as effectively on stridden data as they can work with adjacent data, i.e., with non-stridden vectors. For this computer architectures would have to undergo a dramatic modifications: the memory would have to become multidimensional and the processor architecture would have to become multidimensional too.

3.3 The Householder–QR/QL Algorithm

The Jacobi rotations method is not too bad. It usually converges in between $18N^3$ to $30N^3$ operations. There is no way to get out of the N^3 dependence, but the coefficient can be reduced to $\frac{4}{3}N^3$ (for diagonalization without eigenvectors), and that can be a very considerable saving, especially if N is very large.

The most efficient known technique for finding eigenvalues and eigenvectors of a symmetric matrix is the combination of the Householder reduction, which reduces a symmetric real matrix to a tridiagonal form, followed by the so called QR or a QL algorithm that can diagonalize a tridiagonal matrix within about $30N^2$ steps without eigenvectors. If eigenvectors are required then the number of operations grows to $3N^3$.

The QR or a QL method is still an iterative method. But the orthogonal transformation employed preserves:

- symmetry, and
- tridiagonal form

So zeros stay zeroed. And this means that there are only $N - 2$ off-diagonal elements to kill.

The Householder reduction on the other hand is a finite procedure, i.e., not an iterative one at all. A symmetric matrix can be reduced to a tridiagonal form within a finite well defined number of steps: $N-2$ orthogonal transformations.

3.3.1 The Householder Reduction

The Householder rotation $\boldsymbol{P}$ has the following form:

$$\boldsymbol{P} = \mathbf{1} - 2\boldsymbol{w} \otimes \boldsymbol{w} \tag{3.40}$$

where $\mathbf{1}$ is the Kronecker delta δ_{ij} and $\boldsymbol{w}\otimes\boldsymbol{w}$ is the tensor product, i.e., $\boldsymbol{w}\otimes\boldsymbol{w} \equiv w_i w_j$.

Of vector $\boldsymbol{w}$ we demand only that $\boldsymbol{w} \cdot \boldsymbol{w} = 1$.

Matrix $\boldsymbol{P}$ is clearly symmetric, hence $\boldsymbol{P}^T = \boldsymbol{P}$.

Furthermore matrix $\boldsymbol{P}$ is its own inverse, and this can be seen as follows:

$$\begin{aligned}
\boldsymbol{P} \cdot \boldsymbol{P} &= (\mathbf{1} - 2\boldsymbol{w} \otimes \boldsymbol{w}) \cdot (\mathbf{1} - 2\boldsymbol{w} \otimes \boldsymbol{w}) \\
&= \mathbf{1} \cdot \mathbf{1} - \mathbf{1} \cdot 2\boldsymbol{w} \otimes \boldsymbol{w} - 2\boldsymbol{w} \otimes \boldsymbol{w} \cdot \mathbf{1} + 4\,(\boldsymbol{w} \otimes \boldsymbol{w}) \cdot (\boldsymbol{w} \otimes \boldsymbol{w}) \\
&= \ldots
\end{aligned}$$

Now, observe that

$$(\boldsymbol{w} \otimes \boldsymbol{w}) \cdot (\boldsymbol{w} \otimes \boldsymbol{w}) \equiv \sum_j w_i w_j w_j w_k = w_i w_k \equiv \boldsymbol{w} \otimes \boldsymbol{w}$$

and, of course, $\mathbf{1} \cdot \mathbf{1} = \mathbf{1}$, hence

$$\ldots = \mathbf{1} - 4\boldsymbol{w} \otimes \boldsymbol{w} + 4\boldsymbol{w} \otimes \boldsymbol{w} = \mathbf{1}$$

Consequently: $\boldsymbol{P} = \boldsymbol{P}^T = \boldsymbol{P}^{-1}$, so $\boldsymbol{P}$ is orthogonal.

We can use any vector $\boldsymbol{u}$ in place of $\boldsymbol{w}$ if we normalize it at the same time:

$$\boldsymbol{P} = \mathbf{1} - \frac{2\boldsymbol{u} \otimes \boldsymbol{u}}{\boldsymbol{u} \cdot \boldsymbol{u}} = \mathbf{1} - \frac{\boldsymbol{u} \otimes \boldsymbol{u}}{H} \tag{3.41}$$

where

$$H = \frac{1}{2}\boldsymbol{u} \cdot \boldsymbol{u} \tag{3.42}$$

So far $\boldsymbol{P}$ isn't specific enough to deserve a proud name of a Housholder matrix. But now we choose $\boldsymbol{u}$ to be

$$\boldsymbol{u} = \boldsymbol{x} \mp |\boldsymbol{x}|\boldsymbol{e}_1 \tag{3.43}$$

where $\boldsymbol{e}_1$ is the first basis vector.

The Householder operator with the $\boldsymbol{u}$ vector so defined rotates vector $\boldsymbol{x}$ onto $\boldsymbol{e}_1$, and this can be seen as follows:

$$\begin{aligned} \boldsymbol{P} \cdot \boldsymbol{x} &= \left(\mathbf{1} - \frac{\boldsymbol{u} \otimes \boldsymbol{u}}{H}\right) \cdot \boldsymbol{x} \\ &= \left(\mathbf{1} - \frac{2\boldsymbol{u} \otimes (\boldsymbol{x} \mp |\boldsymbol{x}|\boldsymbol{e}_1)}{\boldsymbol{u} \cdot \boldsymbol{u}}\right) \cdot \boldsymbol{x} \end{aligned} \tag{3.44}$$

What's $\boldsymbol{u} \cdot \boldsymbol{u}$?

$$\begin{aligned} \boldsymbol{u} \cdot \boldsymbol{u} &= (\boldsymbol{x} \mp |\boldsymbol{x}|\boldsymbol{e}_1) \cdot (\boldsymbol{x} \mp |\boldsymbol{x}|\boldsymbol{e}_1) \\ &= \boldsymbol{x} \cdot \boldsymbol{x} \mp 2|\boldsymbol{x}|\boldsymbol{e}_1 \cdot \boldsymbol{x} + |\boldsymbol{x}|^2 \boldsymbol{e}_1 \cdot \boldsymbol{e}_1 \\ &= |\boldsymbol{x}|^2 \mp 2|\boldsymbol{x}|x_1 + |\boldsymbol{x}|^2 \\ &= 2|\boldsymbol{x}|^2 \mp 2|\boldsymbol{x}|x_1 \end{aligned} \tag{3.45}$$

Substituting equation (3.45) into (3.44) yields:

$$\begin{aligned} \boldsymbol{P} \cdot \boldsymbol{x} &= \left(\mathbf{1} - \frac{2\boldsymbol{u} \otimes (\boldsymbol{x} \mp |\boldsymbol{x}|\boldsymbol{e}_1)}{2|\boldsymbol{x}|^2 \mp 2|\boldsymbol{x}|x_1}\right) \cdot \boldsymbol{x} \\ &= \boldsymbol{x} - \frac{2\boldsymbol{u}\left(|\boldsymbol{x}|^2 \mp |\boldsymbol{x}|x_1\right)}{2|\boldsymbol{x}|^2 \mp 2|\boldsymbol{x}|x_1} \\ &= \boldsymbol{x} - \boldsymbol{u} \\ &= \boldsymbol{x} - \boldsymbol{x} \pm |\boldsymbol{x}|\boldsymbol{e}_1 \\ &= \pm|\boldsymbol{x}|\boldsymbol{e}_1 \end{aligned} \tag{3.46}$$

Remember that $\boldsymbol{P}$ is *not* a projection operator. Projection operators are not orthogonal. $\boldsymbol{P}$ is a rotation, which rotates vector $\boldsymbol{x}$ onto the $\boldsymbol{e}_1$ direction. In the process the length of the vector does not change. Whereas it would have changed for a projection operator.

The procedure for transforming matrix $\boldsymbol{A}$ into a tridiagonal form works as follows.

The first Householder operator, $\boldsymbol{P}_1$ is selected to rotate the *sub*-column of the first column:

$$\begin{pmatrix} a_{21} \\ a_{21} \\ \vdots \\ a_{n1} \end{pmatrix}$$

onto

$$\begin{pmatrix} a'_{21} \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

And, to accomplish that, the operator has to have the following form:

$$\boldsymbol{P}_1 = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & & & & \\ \vdots & & & {}^{(n-1)}\boldsymbol{P}_1 & \\ 0 & & & & \end{pmatrix}$$

Multiplying $\boldsymbol{A}$ by $\boldsymbol{P}_1$ from the left:

$$\boldsymbol{P}_1 \cdot \boldsymbol{A} = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & & & & \\ \vdots & & & {}^{(n-1)}\boldsymbol{P}_1 & \\ 0 & & & & \end{pmatrix} \cdot \begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & & & & \\ a_{31} & & & & \\ \vdots & & & & \\ a_{n1} & & & & \end{pmatrix}$$

attacks the first *sub*-column with ${}^{(n-1)}\boldsymbol{P}_1$ and if that *sub*-operator has been chosen to rotate the first sub-column onto its first dimension then the resulting matrix $\boldsymbol{A}'$ will look as follows:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a'_{21} & & & & \\ 0 & & & & \\ \vdots & & & & \\ 0 & & & & \end{pmatrix}$$

Observe that the first row will not be changed by that operation at all, but the lower right corner, of course, will get changed. But now if we apply operator $\boldsymbol{P}_1$ to $\boldsymbol{A}'$ from the right the same thing will happen to the first row (remember that $\boldsymbol{P}$ is symmetric), so that the resulting matrix $\boldsymbol{A}''$ will look as follows:

$$\boldsymbol{P}_1 \cdot \boldsymbol{A} \cdot \boldsymbol{P}_1 = \begin{pmatrix} a_{11} & a'_{12} & 0 & \cdots & 0 \\ a'_{21} & & & & \\ 0 & & & & \\ \vdots & & & & \\ 0 & & & & \end{pmatrix}$$

The second Householder matrix is going to look as follows:

$$\boldsymbol{P}_2 = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & & & \\ \vdots & \vdots & & {}^{(n-2)}\boldsymbol{P}_2 & \\ 0 & 0 & & & \end{pmatrix}$$

and, as you should understand by now, it will do to

$$\begin{pmatrix} a_{32} \\ a_{42} \\ \vdots \\ a_{n2} \end{pmatrix}$$

the same as $\boldsymbol{P}_1$ did to

$$\begin{pmatrix} a_{21} \\ a_{31} \\ a_{41} \\ \vdots \\ a_{n1} \end{pmatrix}$$

In other words it will rotate it onto its first direction, thus leaving only one term under the diagonal. The identity block in the upper left corner ensures that tridiagonalization achieved so far will not be spoiled during successive rotations.

It is now obvious that in $n-2$ such steps the whole matrix must become triadiagonalized.

Because $\boldsymbol{P} = \mathbf{1} - \frac{\boldsymbol{u}\otimes\boldsymbol{u}}{H}$ we can write down our operations on $\boldsymbol{A}$ in more detail. First

$$\boldsymbol{A}\cdot\boldsymbol{P} = \boldsymbol{A}\cdot\left(\mathbf{1} - \frac{\boldsymbol{u}\otimes\boldsymbol{u}}{H}\right) = \boldsymbol{A} - \frac{\boldsymbol{A}\cdot\boldsymbol{u}}{H}\otimes\boldsymbol{u} \tag{3.47}$$

Introducing a new vector $\boldsymbol{p}$:

$$\boldsymbol{p} = \frac{\boldsymbol{A}\cdot\boldsymbol{u}}{H} \tag{3.48}$$

we get

$$\boldsymbol{A}\cdot\boldsymbol{P} = \boldsymbol{A} - \boldsymbol{p}\otimes\boldsymbol{u} \tag{3.49}$$

So now:

$$\begin{aligned} \boldsymbol{P}\cdot\boldsymbol{A}\cdot\boldsymbol{P} &= \left(\mathbf{1} - \frac{\boldsymbol{u}\otimes\boldsymbol{u}}{H}\right)\cdot(\boldsymbol{A} - \boldsymbol{p}\otimes\boldsymbol{u}) \\ &= \boldsymbol{A} - \boldsymbol{p}\otimes\boldsymbol{u} - \boldsymbol{u}\otimes\boldsymbol{p} + \frac{1}{H}\boldsymbol{u}\otimes\boldsymbol{u}\cdot\boldsymbol{p}\otimes\boldsymbol{u} \\ &= \boldsymbol{A} - \boldsymbol{p}\otimes\boldsymbol{u} - \boldsymbol{u}\otimes\boldsymbol{p} + \frac{\boldsymbol{u}\cdot\boldsymbol{p}}{H}\boldsymbol{u}\otimes\boldsymbol{u} \\ &= \boldsymbol{A} - \boldsymbol{p}\otimes\boldsymbol{u} - \boldsymbol{u}\otimes\boldsymbol{p} + 2K\boldsymbol{u}\otimes\boldsymbol{u} \end{aligned} \tag{3.50}$$

where

$$K = \frac{\boldsymbol{u} \cdot \boldsymbol{p}}{2H} \tag{3.51}$$

Our expression for $\boldsymbol{P} \cdot \boldsymbol{A} \cdot \boldsymbol{P}$ can be further simplified by introducing vector $\boldsymbol{q}$:

$$\boldsymbol{q} = \boldsymbol{p} - K\boldsymbol{u} \tag{3.52}$$

and observing that:

$$\boldsymbol{P} \cdot \boldsymbol{A} \cdot \boldsymbol{P} = \boldsymbol{A} - \boldsymbol{q} \otimes \boldsymbol{u} - \boldsymbol{u} \otimes \boldsymbol{q} \tag{3.53}$$

Let us summarise the flow of computation now. For a particular square submatrix of $\boldsymbol{A}$ we'll have

$$\begin{aligned}
\sigma &= a_{21}^2 + a_{31}^2 + a_{41}^2 + \cdots + a_{n1}^2 \\
|\boldsymbol{x}| &= \sqrt{\sigma} \\
\boldsymbol{u} &= \boldsymbol{x} \mp |\boldsymbol{x}|\boldsymbol{e}_1 \\
H &= \frac{\boldsymbol{u} \cdot \boldsymbol{u}}{2} \\
\boldsymbol{p} &= \frac{\boldsymbol{A} \cdot \boldsymbol{u}}{H} \\
K &= \frac{\boldsymbol{u} \cdot \boldsymbol{p}}{2H} \\
\boldsymbol{q} &= \boldsymbol{p} - K\boldsymbol{u} \\
\boldsymbol{A}' &= \boldsymbol{A} - \boldsymbol{q} \otimes \boldsymbol{u} - \boldsymbol{u} \otimes \boldsymbol{q}
\end{aligned}$$

and the accumulated transform, which we are going to need for the eigenvectors is:

$$\boldsymbol{Q} = \boldsymbol{P}_1 \cdot \boldsymbol{P}_2 \cdot \ldots \cdot \boldsymbol{P}_{n-1} \tag{3.54}$$

Of course, you now appreciate that if eigenvectors are needed, this is going to cost us additional operations, because we'll have to compute $\boldsymbol{P} = \boldsymbol{1} - \boldsymbol{u} \otimes \boldsymbol{u}/H$ for every rotation, whereas without eigenvectors we can get away with just the σ, $\boldsymbol{u}$, H, $\boldsymbol{p}$, K, $\boldsymbol{q}$, $\boldsymbol{A}'$ sequence.

Example Code

```
SUBROUTINE tred2(a, d, e, novectors)

  USE nrtype; USE nrutil, ONLY : assert_eq, outerprod
  IMPLICIT NONE

  REAL(sp), DIMENSION(:,:), INTENT(inout) :: a
  REAL(sp), DIMENSION(:), INTENT(out) :: d, e
  LOGICAL(lgt), OPTIONAL, INTENT(in) :: novectors

  INTEGER(i4b) :: i, j, l, n
  REAL(sp) :: f, g, h, hh, scale
  REAL(sp), DIMENSION(SIZE(a, 1)) :: g
  LOGICAL(lgt), SAVE :: yesvec = .TRUE.
  n = assert_eq(SIZE(a, 1), SIZE(a, 2), SIZE(d), SIZE(e), 'tred2')
  IF(PRESENT(novectors)) yesvec = .NOT. novectors
  DO i = n, 2, -1
     l = i-1
     h = 0.0
     IF(l > 1) THEN
        scale = SUM(ABS(a(i, 1:l)))
        IF (scale == 0.0) THEN
           e(i) = a(i, l)
        ELSE
           a(i, 1:l) = a(i, 1:l) / scale
           h = SUM(a(i, 1:l) ** 2)
           f = a(i, l)
           g = -SIGN(SQRT(h), f)
           e(i) = scale*g
           h = h - f*g
           a(i, l) = f - g
           IF (yesvec) a(1:l, i) = a(i, 1:l)/h
           DO j = 1, l
              e(j) = (DOT_PRODUCT(a(j, 1:j), a(i, 1:j)) &
                   + DOT_PRODUCT(a(j+1:l, j), a(i, j+1:l)))/h
           END DO
           f = DOT_PRODUCT(e(1:l), a(i,1:l))
           hh=f/(h+h)
           e(1:l)=e(1:l) - hh*a(i,1:l)
           DO j=1, l
              a(j,1:j) = a(j,1:j)-a(i,j)*e(1:j)-e(j)*a(i,1:j)
           END DO
        END IF
     ELSE
        e(i)=a(i,l)
     END IF
     d(i) = h
  END DO
  IF (yesvec) d(1)=0.0
  e(1) = 0.0
  DO i=1, n
     IF(yesvec) THEN
        l=i-1
        IF (d(i) /= 0.0) THEN
           gg(1:l) = MATMUL(a(i,1:l),a(1:l,1:l))
           a(1:l,1:l) = a(1:l,1:l) - outerprod(a(1:l,i),gg(1:l))
        END IF
        d(i) = a(i, i)
        a(i, i) = 1.0
        a(i, 1:l) = 0.0
        a(1:l, i) = 0.0
     ELSE
        d(i) = a(i, i)
     END IF
  END DO
END SUBROUTINE tred2
```

Subroutine `tred2` performs a Housholder reduction to a triadiagonal form of a real symmetric matrix a, whose dimensions are $n \times n$. On output matrix a is replaced with matrix $\boldsymbol{Q} = \boldsymbol{P}_1 \cdot \boldsymbol{P}_2 \cdot \ldots \boldsymbol{P}_{n-1}$, the diagonal elements are written on array `d` and the $n-1$ off-diagonal elements are written on array `e` with `e(1)=0`. The argument `novectors` is optional:

```
LOGICAL(lgt), OPTIONAL, INTENT(in) :: novectors
```

If it is set to `.TRUE.` then matrix $\boldsymbol{Q}$ is not computed and on exit matrix `a` contains garbage.

You can test for the presence of optional parameters with the keyword `PRESENT`:

```
IF(PRESENT(novectors)) yesvec = .NOT. novectors
```

The action begins by checking dimensions of variables passed to the subroutine as parameters, and if they are correct, i.e., if `a` is $n \times n$, then n is extracted and written on `n`. Otherwise an error message which contains the word `'tred2'` is printed and the subroutine exits.

The main loop has the form

```
DO i = n, 2, -1
   l = i-1
   h = 0.0
   IF(l > 1) THEN
      blah... blah... blah...
   ELSE
      e(i)=a(i,l)
   END IF
   d(i) = h
END DO
```

The first thing to observe is that in this program we're moving not from the upper left corner towards the lower right corner, but the other way round, i.e., our index `i` starts at the last column number `n`, and then moves down towards the beginning of the matrix. In other words our first evaluation will yield a new value for $a_{n-1,n}$ or $a_{n,n-1}$, depending on whether we're going to work on the lower or upper triangular part of the matrix. Remember that as we move towards the end, our operators ${}^{(n-k)}\boldsymbol{P}_k$ become smaller and smaller, until the last one is simply the identity, which is why for $i = 2$ and $l = 1$ we have:

```
      e(i)=a(i,l)
```

But note that by this stage `a(i,l)` is going to be something quite different from what it was at the beginning. The first element of array `e` is set to `0` after the main `DO` loop exits following the `IF` statement:

```
IF (yesvec) d(1)=0.0
e(1) = 0.0
```

Now let us have a look at the `blah... blah... blah...` clause in the main `DO` loop. What we find inside there is another `IF` statement:

```
      scale = SUM(ABS(a(i, 1:l)))
      IF (scale == 0.0) THEN
         e(i) = a(i, l)
```

```
ELSE
   blah... blah... blah...
END IF
```

First we evaluate $\sum_{k=1}^{i-1} |a_{i,k}|$, i.e., we sum absolute values of all elements in row i up to, but excluding, the diagonal element $a_{i,i}$. If that whole row adds up to zero, there is no point zeroing it further, so we simply transfer $a_{i,i-1}$ to `e(i)` and go on to rotate the next row out of existence.

At this stage we have exhausted all possible avenues for avoidance behaviour and have to do some real work, i.e., have to fill the `blah... blah... blah...` bit above.

The first thing we do is to scale $a_{i,1}, a_{i,2}, \ldots, a_{i,i-1}$:

```
a(i, 1:l) = a(i, 1:l) / scale
```

This we do for the sake of improving numerical accuracy and stability. Is this kosher though? Remember that

$$\boldsymbol{A}' = \boldsymbol{A} - \boldsymbol{q} \odot \boldsymbol{u} \tag{3.55}$$

where $\odot$ is a symmetric tensor product, and $\boldsymbol{q} = \boldsymbol{p} - K\boldsymbol{u}$. Now those vectors $\boldsymbol{p}$ and $\boldsymbol{q}$ are defined by dividing various combinations of $\boldsymbol{u}$ by H, which is $\boldsymbol{u} \cdot \boldsymbol{u}$. Consequently $\boldsymbol{q} \odot \boldsymbol{u}$ eventually divides $\boldsymbol{u} \otimes \boldsymbol{u}$ by $\boldsymbol{u} \cdot \boldsymbol{u}$, which implies that it is OK to scale a column or a row before rotating it, because $\boldsymbol{u} = \boldsymbol{x} \mp |\boldsymbol{x}|\boldsymbol{e}_1$.

The next step evaluates what we have called σ, i.e., $\sigma = a_{i,1}^2 + a_{i,2}^2 + \ldots + a_{i,i-1}^2 = \boldsymbol{x} \cdot \boldsymbol{x}$:

```
h = SUM(a(i, 1:l) ** 2)
```

Now we calculate $\mp|\boldsymbol{x}|$ and the sign we choose is the opposite to the sign of $a_{i,i-1}$:

```
f = a(i, l)
g = -SIGN(SQRT(h), f)
```

The off-diagonal element that we're going to save on the array `e` is now going to be that `g` scaled back up to the original value. Remember we have shown that $\boldsymbol{P} \cdot \boldsymbol{x} = \pm|\boldsymbol{x}|\boldsymbol{e}_1$:

```
e(i) = scale*g
```

Now recall that $\boldsymbol{u} \cdot \boldsymbol{u} = 2|\boldsymbol{x}|^2 \mp 2|\boldsymbol{x}|x_1$. This is implemented in the code as:

```
h = h - f*g
```

Consequently, now `h` becomes H, whereas previously it was σ.

Vector $\boldsymbol{u}$ is equal to the original vector $\boldsymbol{x}$ everywhere with the exception of its head. So, if we just modify the latter, we'll have $\boldsymbol{u}$ stored in the i^{th} row of matrix `a`:

```
a(i, l) = f - g
```

where f is $a_{i,i-1}$ and g is $\mp|\boldsymbol{x}|x_1$.

At the same time we store $\boldsymbol{u}/H$ is the i^{th} column of matrix a for future use in case the operator $\boldsymbol{Q}$ is to be evaluated:

```
IF (yesvec) a(1:l, i) = a(i, 1:l)/h
```

The next step is to evaluate $\boldsymbol{p} = \boldsymbol{A} \cdot \boldsymbol{u}/H$. This is done by the following code fragment:

```
DO j = 1, l
   e(j) = (DOT_PRODUCT(a(j, 1:j), a(i, 1:j)) &
        + DOT_PRODUCT(a(j+1:l, j), a(i, j+1:l)))/h
END DO
```

The reason for such equilibristics is that by now we have already corrupted matrix $\boldsymbol{A}$ and we must carefully pick up the right fragments of it in order to perform this multiplication. Remember that $\boldsymbol{u}$ is stored in row i. We store this result on an *unused* part of array e.

The next two lines evaluate $K = \boldsymbol{u} \cdot \boldsymbol{p}/(2H)$:

```
f = DOT_PRODUCT(e(1:l), a(i,1:l))
hh=f/(h+h)
```

And once we have K we can evaluate vector $\boldsymbol{q} = \boldsymbol{p} - K\boldsymbol{u}$:

```
e(1:l)=e(1:l) - hh*a(i,1:l)
```

And the result is once again stored on the unused portion of array e. We won't need vector $\boldsymbol{p}$ any more so we can overwrite it with $\boldsymbol{q}$.

Finally we are ready to perform the final $\boldsymbol{A}' = \boldsymbol{A} - \boldsymbol{q} \otimes \boldsymbol{u} - \boldsymbol{u} \otimes \boldsymbol{q}$:

```
DO j=1, l
   a(j,1:j) = a(j,1:j)-a(i,j)*e(1:j)-e(j)*a(i,1:j)
END DO
```

Remember that we have promised to return the diagonal elements on array d, but so far we haven't made any use of that array. Luckily, for the time being the new diagonal elements are stored safely on $a_{i,i}$. If matrix $\boldsymbol{Q}$ is not wanted, then we're done and we return with just that:

```
e(1) = 0.0
DO i=1, n
   IF(yesvec) THEN
      blah... blah... blah...
   ELSE
      d(i) = a(i, i)
   END IF
END DO
```

If $\boldsymbol{Q}$ is wanted though, then we make use of vectors $\boldsymbol{u}$ and $\boldsymbol{u}/H$, which are now stored on rows and on columns of matrix a. The operation $\boldsymbol{P}$ has the following form: $\boldsymbol{P} = \mathbf{1} - \boldsymbol{u} \otimes \boldsymbol{u}/H$. We continue to use matrix a as a working storage.

The logic of this part of the code is as follows:

```
d(1)=0.0
DO i=1, n
   l=i-1
   IF (d(i) /= 0.0) THEN
      gg(1:l) = MATMUL(a(i,1:l),a(1:l,1:l))
      a(1:l,1:l) = a(1:l,1:l) - outerprod(a(1:l,i),gg(1:l))
   END IF
   blah... blah... blah...
END DO
```

Remember that before we got to this place `d(i)` was set to `h`, which, in turn, was set to H. Setting `d(1)=0` and then using the condition `IF(d(i) /= 0.0)` protects us from ever reaching for `a(0,...)`. So the computation within the `IF` statement is performed only for $i \geq 2$.

The computation, in turn, reflects the following recursive relation:

$$
\begin{aligned}
\boldsymbol{Q} &= \boldsymbol{P}_1 \cdot \boldsymbol{P}_2 \cdot \ldots \cdot \boldsymbol{P}_{n-2} \\
\boldsymbol{Q}_{n-2} &= \boldsymbol{P}_{n-2} \\
\boldsymbol{Q}_j &= \boldsymbol{P}_j \cdot \boldsymbol{Q}_{j+1} = \left(\mathbf{1} - \frac{\boldsymbol{u}_j \otimes \boldsymbol{u}_j}{H_j}\right) \cdot \boldsymbol{Q}_{j+1} \\
&= \boldsymbol{Q}_{j+1} - \boldsymbol{u}_j \otimes \frac{\boldsymbol{u}_j}{H} \cdot \boldsymbol{Q}_{j+1} \\
\boldsymbol{Q} &= \boldsymbol{Q}_1
\end{aligned}
$$

Thus the first statement of the code in the `IF` brackets simply calculates $\frac{\boldsymbol{u}_j}{H} \cdot \boldsymbol{Q}_{j+1}$ and then the second statement evaluates $\boldsymbol{Q}_{j+1} - \boldsymbol{u}_j \otimes$ (first statement).

Now we clean up for the next iteration:

```
d(i) = a(i, i)
a(i, i) = 1.0
a(i, 1:l) = 0.0
a(1:l, i) = 0.0
```

The diagonal element is finally transferred to array d, and the corner portion of $\boldsymbol{A}$ that should be set to an identity matrix is set to an identity matrix: remember that the full matrix $\boldsymbol{P}$ comprised a smaller matrix ${}^{(n-k)}\boldsymbol{P}_k$ and an identity matrix in the opposite corner.

3.3.2 A Little Summary

Let us summarise what we have learnt about Jacobi and Householder rotations.

Jacobi rotations are designed to annihilate a selected off-diagonal element, say, a_{pq}. Given matrix elements a_{qq}, a_{pp}, and a_{pq} the rotation itself can be calculated as follows:

$$\begin{aligned}
\theta &= \cot 2\varphi = \frac{a_{qq} - a_{pp}}{2a_{pq}} \\
t &= \tan\varphi = \frac{\operatorname{sign}(\theta)}{|\theta| + \sqrt{\theta^2 + 1}} \\
c &= \cos\varphi = \frac{1}{\sqrt{t^2 + 1}} \\
s &= \sin\varphi = ct \\
\tau &= \tan\frac{\varphi}{2} = \frac{s}{1 + c} \\
a'_{pp} &= a_{pp} - ta_{pq} \\
a'_{qq} &= a_{qq} + ta_{pq} \\
a'_{rp} &= a_{rp} - s\left(a_{rq} + \tau a_{rp}\right) \\
a'_{rq} &= a_{rq} + s\left(a_{rp} - \tau a_{rq}\right) \\
v'_{rp} &= v_{rp} - s\left(v_{rq} + \tau v_{rp}\right) \\
v'_{rq} &= v_{rq} + s\left(v_{rp} - \tau v_{rq}\right)
\end{aligned}$$

Householder rotations are designed to rotate the whole under diagonal column or the whole right-to-diagonal row onto its first direction. Given a subdiagonal column:

$$\boldsymbol{x} = \begin{pmatrix} a_{p+1,p} \\ a_{p+2,p} \\ \vdots \\ a_{n,p} \end{pmatrix}$$

the rotation itself can be calculated as follows:

$$\begin{aligned}
\sigma &= \boldsymbol{x} \cdot \boldsymbol{x} \\
|\boldsymbol{x}| &= \sqrt{\sigma} \\
\boldsymbol{u} &= \boldsymbol{x} \mp |\boldsymbol{x}|\boldsymbol{e}_1 \\
H &= \frac{\boldsymbol{u} \cdot \boldsymbol{u}}{2} \\
\boldsymbol{p} &= \frac{\boldsymbol{A} \cdot \boldsymbol{u}}{H} \\
K &= \frac{\boldsymbol{u} \cdot \boldsymbol{p}}{2H} \\
\boldsymbol{q} &= \boldsymbol{p} - K\boldsymbol{u} \\
\boldsymbol{A}' &= \boldsymbol{A} - \boldsymbol{q} \otimes \boldsymbol{u} - \boldsymbol{u} \otimes \boldsymbol{q} \\
\boldsymbol{Q} &= \boldsymbol{P}_1 \cdot \boldsymbol{P}_2 \cdot \ldots \boldsymbol{P}_{n-2} \\
\boldsymbol{Q}_{n-2} &= \boldsymbol{P}_{n-2} \\
\boldsymbol{Q}_j &= \boldsymbol{Q}_{j+1} - \boldsymbol{u}_j \otimes \frac{\boldsymbol{u}_j}{H} \cdot \boldsymbol{Q}_{j+1} \\
\boldsymbol{Q} &= \boldsymbol{Q}_1
\end{aligned}$$

3.3.3 Givens Rotations

There is another type of rotations, which are called *Givens* rotations. These are like Jacobi rotations, but their purpose is not to annihilate one of the corner elements, a_{pp}, a_{qq}, a_{qp}, or a_{pq}, but instead to annihilate elements in the top row, i.e.,

$$\begin{array}{lll} \boldsymbol{P}_{23} & \text{annihilates} & a_{31} \text{ and } a_{13} \\ \boldsymbol{P}_{24} & \text{annihilates} & a_{41} \text{ and } a_{14} \\ \boldsymbol{P}_{25} & \text{annihilates} & a_{51} \text{ and } a_{15} \end{array}$$

and so on.

Because $a'_{rp} = a_{rp} - s\,(a_{rq} + \tau a_{rp})$ and $a'_{rq} = a_{rq} + s\,(a_{rp} - \tau a_{rq})$ for $r \neq p$ and $r \neq q$, if a_{rp} and a_{rq} have been set to zero they will *remain* zero. So these Givens rotations are not unlike Householder rotations, but for normal filled matrices they are somewhat less efficient. However, they're actually a little more lightweight for tridiagonal matrices than Householder rotations, so we'll use them to effect the final diagonalisation.

3.3.4 The QR and QL Algorithms

Any real matrix, symmetric or not, can be decomposed into either of the forms:

$$\boldsymbol{A} = \boldsymbol{Q}\cdot\boldsymbol{R}, \text{ or} \tag{3.56}$$

$$\boldsymbol{A} = \boldsymbol{Q}\cdot\boldsymbol{L} \tag{3.57}$$

where $\boldsymbol{R}$ is the upper triangular matrix (the *right* part) that includes the diagonal, $\boldsymbol{L}$ is the lower triangular matrix (the *left* part) that includes the diagonal, and $\boldsymbol{Q}$ is a rotation matrix.

It is easy to see how such a decomposition can be eventuated by applying, for example, Householder rotations or Givens rotations to matrix $\boldsymbol{A}$, in order to annihilate everything below the diagonal. Observe that these will not be full similarity transformations, in other words they are not going to be $\boldsymbol{P}^T\cdot\boldsymbol{A}\cdot\boldsymbol{P}$. Remember that we have acted with $\boldsymbol{P}$ on $\boldsymbol{P}^T\cdot\boldsymbol{A}$ from the right in order to do to rows, what application of $\boldsymbol{P}^T$ to $\boldsymbol{A}$ from the left did to the columns. So if we just leave $\boldsymbol{P}^T\cdot\boldsymbol{A}$ alone, we'll be left with a new matrix $\tilde{\boldsymbol{A}}$, with, say, k-th column zeroed under the diagonal. An accumulation of those $\boldsymbol{P}_i^T$ is some combined rotation transformation $\boldsymbol{Q}^T$, such that

$$\boldsymbol{Q}^T\cdot\boldsymbol{A} = \boldsymbol{R}$$

Multiplying by $\boldsymbol{Q}$ from the left yields:

$$\boldsymbol{A} = \boldsymbol{Q}\cdot\boldsymbol{R}$$

quod erat demonstrandum.

Now, once we have done all that hard work and found both $\boldsymbol{Q}$ and $\boldsymbol{R}$, by merely changing the order, i.e., replacing $\boldsymbol{Q} \cdot \boldsymbol{R}$ with $\boldsymbol{R} \cdot \boldsymbol{Q}$ we eventuate a similarity transformation on $\boldsymbol{A}$:

$$\boldsymbol{R} \cdot \boldsymbol{Q} = \boldsymbol{Q}^T \cdot \boldsymbol{A} \cdot \boldsymbol{Q} = \boldsymbol{A}'$$

This is called the $\boldsymbol{QR}$ transformation of matrix $\boldsymbol{A}$ and it has the following properties:

1. it preserves symmetry of $\boldsymbol{A}$
2. it preserves triadiagonal form of $\boldsymbol{A}$
3. it preserves Hessenberg form of $\boldsymbol{A}$

A *Hessenberg* matrix is a matrix with everything under the diagonal and under the one line that is under the diagonal (like in a tridiagonal matrix) set to zero.

The $\boldsymbol{QR}$ algorithm now works as follows. Once you have a tridiagonal matrix $\boldsymbol{A}$, you

1. find its $\boldsymbol{Q} \cdot \boldsymbol{R}$ decomposition
2. generate $\boldsymbol{A}_1 = \boldsymbol{R} \cdot \boldsymbol{Q}$
3. find the $\boldsymbol{Q}_1 \cdot \boldsymbol{R}_1$ decomposition of $\boldsymbol{A}_1$
4. generate $\boldsymbol{A}_2 = \boldsymbol{R}_1 \cdot \boldsymbol{Q}_1$
5. find the $\boldsymbol{Q}_2 \cdot \boldsymbol{R}_2$ decomposition of $\boldsymbol{A}_2$
6. generate $\boldsymbol{A}_3 = \boldsymbol{R}_2 \cdot \boldsymbol{Q}_2$
7. ...

and so on, until the off-diagonal elements vanish. Now, will they vanish? Will this whole process really end up in diagonalization of matrix $\boldsymbol{A}$?

theorem

1. If a general real, not necessarily symmetric matrix $\boldsymbol{A}$ has eigenvalues of different absolute value $|\lambda_i|$ then the sequence $\boldsymbol{A}_i$, as defined by the $\boldsymbol{QR}$ transformations of $\boldsymbol{A}$, converges to an upper triangular form, with eigenvalues of $\boldsymbol{A}$ appearing on the diagonal in increasing order of their absolute magnitude.
2. If $\boldsymbol{A}$ has an eigenvalue $|\lambda_i|$ of multiplicity p then $\boldsymbol{A}_i$ still converges to an upper triangular form with the exception for a diagonal block matrix of order p whose eigenvalues are λ_i.

The workload for this algorithm is $\mathcal{O}(n^3)$ per iteration for a general matrix, but only $\mathcal{O}(n)$ per iteration for a tridiagonal matrix, and $\mathcal{O}(n^2)$ for a Hessenberg matrix.

> *Which is why the combination of the Householder triadiagonalization followed by the **QR** transformations is such an efficient method for finding eigenvalues of a symmetric matrix.*

If there is a degenerate eigenvalue, i.e., a value $|\lambda_i|$ of multiplicity p, then at the end of this procedure there will be $p-1$ non-vanishing elements on the super and sub-diagonal, that correspond to that value. Those elements then determine a submatrix that can be cut out of the original matrix $\boldsymbol{A}$ and diagonalized separately, for example, by evaluating a characteristic polynomial, or by Jacobi iterations.

There is nothing sacred about $\boldsymbol{QR}$ versus $\boldsymbol{QL}$, so everything that's been said about $\boldsymbol{QR}$ works just as well for $\boldsymbol{QL}$. But when you write a program, of course, you must choose one or the other. Since academics have a long history of leaning towards the left, we're going to choose $\boldsymbol{QL}$ in our example code too.

Shifting

Remember that the equation:

$$\boldsymbol{A} \cdot \boldsymbol{x} = \lambda \boldsymbol{x}$$

defines λ_i up to a constant. That is, you can add the same constant to all λ_i and still have the same $\boldsymbol{x}_i$ solving the equation above. In physics, for example, this translates into the statement that energy is defined up to an additive constant, or that there is no absolute zero for energy. And energy levels in Quantum Mechanics are eigenvalues of the Hamiltonian.

The rate with which off-diagonal elements converge to zero in the $\boldsymbol{QL}$ sequence is given by

$$a_{ij}^{(s)} \sim \left(\frac{\lambda_i}{\lambda_j}\right)^s$$

which explains why it is so hard to kill off-diagonal terms that correspond to degenerate eigenvalues. If two eigenvalues λ_i and λ_j are very close then convergence can be slow. This convergence can be accelerated by decomposing:

$$\boldsymbol{A}_s - k_s \boldsymbol{1} = \boldsymbol{Q}_s \cdot \boldsymbol{L}_s$$

instead of $\boldsymbol{A}_s$. The $\boldsymbol{QL}$ transformation of this equation now yields:

$$\boldsymbol{L}_s \cdot \boldsymbol{Q}_s = \boldsymbol{Q}_s^T \cdot (\boldsymbol{A}_s - k_s \boldsymbol{1}) \cdot \boldsymbol{Q}_s = \boldsymbol{Q}_s^T \cdot \boldsymbol{A}_s \cdot \boldsymbol{Q}_s - k_s \boldsymbol{1} = \boldsymbol{A}_{s+1} - k_s \boldsymbol{1}$$

So, we can always reconstruct $\boldsymbol{A}_{s+1}$ by performing

$$\boldsymbol{A}_{s+1} = \boldsymbol{L}_s \cdot \boldsymbol{Q}_s + k_s \boldsymbol{1}$$

But the convergence in this procedure will be given by

$$\frac{\lambda_i - k_s}{\lambda_j - k_s}$$

and by choosing $k_s \approx \lambda_j$ we can, in principle, speed it up enormously.

But the difficulty is that we don't know what λ_j is going to be!

> *A common strategy is to compute the eigenvalues of the leading* 2×2 *diagonal submatrix of* $\boldsymbol{A}$ *and then set* k_s *to the* λ_i *that is closer to* a_{11}.

One can show that the convergence here is cubic or at worst quadratic if eigenvalues are degenerate.

Although in general the $\boldsymbol{QL}$ decomposition is obtained by a sequence of Householder transformations, for a tridiagonal matrix Jacobi rotations $\boldsymbol{P}_{pq}$ can be used and are cheaper. A sequence of

$$\boldsymbol{P}_{12}, \boldsymbol{P}_{23}, \boldsymbol{P}_{34}, \ldots$$

will eliminate

$$a_{12}, a_{23}, a_{34}, \ldots$$

and

$$a_{21}, a_{32}, a_{43}, \ldots$$

The resulting $\boldsymbol{Q}^T$ will therefore be:

$$\boldsymbol{Q}_s^T = \boldsymbol{P}_{1,2}^{(s)} \cdot \boldsymbol{P}_{2,3}^{(s)} \cdot \boldsymbol{P}_{3,4}^{(s)} \cdot \ldots \cdot \boldsymbol{P}_{n-1,n}^{(s)}$$

QL Algorithm with Implicit Shifts

In this algorithm we begin by applying a Jacobi rotation, from the left hand side only, whose purpose is to annihilate $A_{n-1,n}$, i.e., the last superdiagonal element.

This rotation will usually screw the tridiagonal form, so the tridiagonal form must be restored either by using a Housholder or a Givens transformation. It turns out that a single Givens transformation won't do though: it must be followed by a serious of those, so as to comb away the non-tridiagonal terms. A single iteration $\bar{\boldsymbol{Q}}^T$ will thus have a form:

$$\bar{\boldsymbol{Q}}_s^T = \bar{\boldsymbol{P}}_1^{(s)} \cdot \bar{\boldsymbol{P}}_2^{(s)} \cdot \bar{\boldsymbol{P}}_3^{(s)} \cdot \ldots \cdot \bar{\boldsymbol{P}}_{n-2}^{(s)} \cdot \boldsymbol{P}_{n-1}^{(s)} \qquad (3.58)$$

where $\boldsymbol{P}_{n-1}^{(s)}$ is the corresponding Jacobi rotation and $\bar{\boldsymbol{P}}_{n-2}^{(s)}$ (with a bar) are Givens transformations that re-tridiagonalize the matrix.

Now, the question is if this transformation $\boldsymbol{Q}$ is going to be a correct $\boldsymbol{Q}$ for a $\boldsymbol{QL}$ iteration. There is a tricky lemma, which proves that this is indeed the case:

lemma

> If $\boldsymbol{A}$ is a symmetric nonsingular matrix and $\boldsymbol{B} = \boldsymbol{Q}^T \cdot \boldsymbol{A} \cdot \boldsymbol{Q}$, where $\boldsymbol{Q}$ is orthogonal and $\boldsymbol{B}$ is tridiagonal, not necessarily symmetric, with positive off-diagonal terms, then $\boldsymbol{Q}$ and $\boldsymbol{B}$ are fully determined by the last row of $\boldsymbol{Q}^T$.

proof

Let $\boldsymbol{q}_1$ through $\boldsymbol{q}_n$ be the rows of matrix $\boldsymbol{Q}^T$:

$$\boldsymbol{Q}^T = \begin{pmatrix} \boldsymbol{q}_1 \\ \boldsymbol{q}_2 \\ \vdots \\ \boldsymbol{q}_n \end{pmatrix}$$

Since $\boldsymbol{Q}$ is orthogonal, $\boldsymbol{B} = \boldsymbol{Q}^T \cdot \boldsymbol{A} \cdot \boldsymbol{Q}$ can be multiplied by $\boldsymbol{Q}^T$ from the right yielding:

$$\boldsymbol{B} \cdot \boldsymbol{Q}^T = \boldsymbol{Q}^T \cdot \boldsymbol{A}$$

This can be written down in some detail as

$$\begin{pmatrix} \beta_1 & \gamma_1 & & & & \\ \alpha_2 & \beta_2 & \gamma_2 & & & \\ & & \vdots & & & \\ & & & \alpha_{n-1} & \beta_{n-1} & \gamma_{n-1} \\ & & & & \alpha_n & \beta_n \end{pmatrix} \cdot \begin{pmatrix} \boldsymbol{q}_1 \\ \boldsymbol{q}_2 \\ \vdots \\ \boldsymbol{q}_{n-1} \\ \boldsymbol{q}_n \end{pmatrix} = \begin{pmatrix} \boldsymbol{q}_1 \\ \boldsymbol{q}_2 \\ \vdots \\ \boldsymbol{q}_{n-1} \\ \boldsymbol{q}_n \end{pmatrix} \cdot \boldsymbol{A}$$

The n^{th} row of this equation can be written as

$$\alpha_n \boldsymbol{q}_{n-1} + \beta_n \boldsymbol{q}_n = \boldsymbol{q}_n \cdot \boldsymbol{A}$$

The rows $\boldsymbol{q}_i$ of matrix $\boldsymbol{Q}$ are orthonormal to each other, so if we multiply the above equation from the right by $\boldsymbol{q}_n^T$, we get:

$$\beta_n = \boldsymbol{q}_n \cdot \boldsymbol{A} \cdot \boldsymbol{q}_n^T$$

Once we know β_n we can evaluate α_n:

$$\alpha_n \boldsymbol{q}_{n-1} = \boldsymbol{q}_n \cdot \boldsymbol{A} - \beta_n \boldsymbol{q}_n = \boldsymbol{z}_{n-1}$$

Using orthonormality of $\boldsymbol{q}_i$:

$$\alpha_n^2 = \boldsymbol{z}_{n-1} \cdot \boldsymbol{z}_{n-1}$$

and

$$\alpha_n = |\boldsymbol{z}_{n-1}|$$

And, therefore

$$\boldsymbol{q}_{n-1} = \boldsymbol{z}_{n-1} / \alpha_n$$

We can now climb up towards β_1 and γ_1 reconstructing the whole $\boldsymbol{B}$ and $\boldsymbol{Q}$ in this way.

Quod erat demonstrandum.

The lemma is used by observing that equation:

$$\boldsymbol{Q}_s^T = \boldsymbol{P}_{1,2}^{(s)} \cdot \boldsymbol{P}_{2,3}^{(s)} \cdot \boldsymbol{P}_{3,4}^{(s)} \cdot \ldots \cdot \boldsymbol{P}_{n-1,n}^{(s)}$$

which defines the iterative procedure for the $\boldsymbol{QL}$ method implies that the last row of $\bar{\boldsymbol{Q}}^T$, which is determined solely by $\boldsymbol{P}_{n-1,n}^{(s)}$, is the same as the last row of $\boldsymbol{Q}_s^T$.

Once we have $\boldsymbol{Q}_s^T$ we form the next iterate $\boldsymbol{A}_{s+1}$ by:

$$\boldsymbol{A}_{s+1} = \boldsymbol{Q}_s^T \cdot \boldsymbol{A}_s \cdot \boldsymbol{Q}_s$$

In this method not only are shifts implicit. The $\boldsymbol{QL}$ algorithm itself is masked. We rely on the lemma to demonstrate that the computation, which at first glance looks quite different, is equivalent to a $\boldsymbol{QL}$ algorithm.

Example Code

```
SUBROUTINE tqli(d,e,z)

  USE nrtype; USE nrutil, ONLY : assert_eq, nrerror
  USE nr, ONLY : pythag

  IMPLICIT NONE

  REAL(sp), DIMENSION(:), INTENT(inout) :: d, e
  REAL(sp), DIMENSION(:,:), OPTIONAL, INTENT(inout) :: z

  INTEGER(i4b) :: i, iter, l, m, n, ndum
  REAL(sp) :: b, c, dd, f, g, p, r, s
  REAL(sp), DIMENSION(SIZE(e)) :: ff

  n = assert_eq(SIZE(d), SIZE(e), 'tqli: n')
  IF(PRESENT(z)) ndum=assert_eq(n, SIZE(z, 1), SIZE(z, 2), 'tqli: ndum')
  e(:)=EOSHIFT(e(:),1)
  DO l=1, n
     iter=0
     iterate: DO
        DO m=l, n-1
           dd=ABS(d(m)) + ABS(d(m+1))
           IF(ABS(e(m))+dd == dd) EXIT
        END DO
        IF (m==l) EXIT iterate
        IF(iter == 30) CALL nrerror('too many iterations in tqli')
        iter = iter+1
        g=(d(l+1)-d(l))/(2.0_sp*e(l))
        r = pythag(g, 1.0_sp)
        g = d(m) - d(l) + e(l) / (g + SIGN(r, g))
        s = 1.0
        c = 1.0
        p = 0.0
        DO i = m-1, l, -1
           f = s*e(i)
           b = c*e(i)
           r = pythag(f, g)
```

```
          e(i+1) = r
          IF(r == 0.0) THEN
             d(i+1) = d(i+1) - p
             e(m) = 0.0
             CYCLE iterate
          END IF
          s = f/r
          c = g/r
          g = d(i + 1) - p
          r = (d(i) - g) * s + 2.0_sp * c * b
          p = s * r
          d(i + 1) = g + p
          g = c * r - b
          IF(PRESENT(z)) THEN
             ff(1:n) = z(1:n,i+1)
             z(1:n, i+1) = s*z(1:n,i) + c*ff(1:n)
             z(1:n,i) = c*z(1:n, i) - s*ff(1:n)
          END IF
       END DO
       d(l) = d(l) - p
       e(l) = g
       e(m) = 0.0
    END DO iterate
  END DO
END SUBROUTINE tqli
```

This subroutine takes two arrays of length n on input: `d`, which contains diagonal terms of a tridiagonal symmetric matrix $\boldsymbol{A}$, and `e`, which contains sub- (and super-) diagonal terms of $\boldsymbol{A}$. The third parameter, `z`, is optional. If it is provided the eigenvectors that correspond to the diagonalising transformation $\boldsymbol{Q}$ (and thus the transformation itself) are written on it. Its dimensions must therefore be $n \times n$. On output the subroutine returns diagonal terms on `d`, whereas vector `e` is destroyed. Because there are only $n - 1$ sub-diagonal terms, `e(1)` is ignored and the valid terms are assumed to live on `e(2)` through `e(n)`.

The first two statements check if arrays `d` and `e` are properly dimensioned. If they are the dimension is written on `n`, if not, an error message is written on standard output, which contains a string `'tqli: n'` and the subroutine aborts. Then, if the third argument `z` is present, we check if that matrix is correctly dimensioned, i.e., $n \times n$ and write the dimension on `ndum`, which is a dummy argument that's not used by the subroutine any more, because the real n already lives in `n`:

```
  n = assert_eq(SIZE(d), SIZE(e), 'tqli: n')
  IF(PRESENT(z)) ndum=assert_eq(n, SIZE(z, 1), SIZE(z, 2), 'tqli: ndum')
```

The next step is to shift array `e` to the left by one place for convenience:

```
  e(:)=EOSHIFT(e(:),1)
```

so that the first valid element of it goes into `e(1)` and the last goes into `e(n-1)`, i.e., the matrix $\boldsymbol{A}$ now looks as follows:

$$\begin{pmatrix} d_1 & e_1 & 0 & 0 & \dots & 0 & 0 \\ e_1 & d_2 & e_2 & 0 & \dots & 0 & 0 \\ 0 & e_2 & d_3 & e_3 & \dots & 0 & 0 \\ 0 & 0 & e_3 & d_4 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & d_n & e_{n-1} \\ 0 & 0 & 0 & 0 & \dots & e_{n-1} & d_n \end{pmatrix}$$

Now the real fun begins. The main body of the subroutine is a large `DO` loop, with another `DO` loop inside and a number of conditionals:

```
DO l=1, n
   iter=0
   iterate: DO
      DO m=l, n-1
         dd=ABS(d(m)) + ABS(d(m+1))
         IF(ABS(e(m))+dd == dd) EXIT
      END DO
      IF (m==l) EXIT iterate
      IF(iter == 30) CALL nrerror('too many iterations in tqli')
      iter = iter+1

      blah... blah... blah...

      DO i = m-1, l, -1

         blah... blah... blah...

         IF(r == 0.0) THEN
            d(i+1) = d(i+1) - p
            e(m) = 0.0
            CYCLE iterate
         END IF

         blah... blah... blah...

      END DO

      blah... blah... blah...

      d(l) = d(l) - p
      e(l) = g
      e(m) = 0.0
   END DO iterate
END DO
```

There are several ideas here. First we move downwards, i.e., we start with $l = 1$, diagonalize that in some way, then we *deflate* matrix $\boldsymbol{A}$ by crossing out the first row and the first column and repeat the same operation to a smaller matrix, and we keep doing that until the whole $\boldsymbol{A}$ is done.

As you remember the $\boldsymbol{QL}$ algorithm generates eigenvalues in the order of diminishing absolute value. So once the first such value pops up in the upper left corner, that part of the job is finished, and the matrix can be deflated.

Within each such deflated submatrix, as l moves towards n, we try to *split* the submatrix of $\boldsymbol{A}$ on a very small subdiagonal element e_m, so small that

$$|d_m| + |d_{m+1}| + |e_m| = |d_m| + |d_{m+1}|$$

within the accuracy of floating point operations. Assuming that we have found such an element, we are eventually going to set it to zero formally (at the end of the `iterate` loop), while performing some mathematics on the elements between m and l. If we hit on such an element at the very beginning, i.e., if $m = l$, then it means that for this l the off-diagonal term is already negligible and so we can move to $l + 1$ right away. If we don't find such an element at all, then m will eventually become $n - 1$, i.e., we'll just have to deal with the whole submatrix, that is a *deflated* $\boldsymbol{A}$.

Now, recall the shifting strategy:

> *Choose k_s equal to the eigenvalue of the leading 2×2 submatrix that is closer to d_l.*

OK, so we have this little submatrix:

$$\begin{pmatrix} d_l & e_l \\ e_l & d_{l+1} \end{pmatrix}$$

and how do we find its eigenvalue? Well, we simply hit it with the Jacobi rotation:

$$\begin{aligned} \theta &= \frac{d_{l+1} - d_l}{2e_l} \\ t &= \frac{\operatorname{sign}(\theta)}{|\theta| + \sqrt{\theta^2 + 1}} \\ d_l' &= d_l - te_l = d_l - \frac{e_l \operatorname{sign}(\theta)}{|\theta| + \sqrt{\theta^2 + 1}} \end{aligned}$$

and this d_l' is now going to be our k_s, i.e., the shift.

In the code this is implemented as:

```
g=(d(l+1)-d(l))/(2.0_sp*e(l))
r = pythag(g, 1.0_sp)
g = d(m) - d(l) + e(l) / (g + SIGN(r, g))
```

where g is initially θ, r is $\sqrt{\theta^2 + 1}$, the shift $k_s = d_l'$ becomes `d(l) - e(l) / (g + SIGN(r, g))`, and the second instance of g becomes $d_m - k_s$.

Now let us have a look at the lower right corner of our submatrix of a deflated original $\boldsymbol{A}$ that results from a successful splitting:

$$\begin{array}{llllll}
\ddots & e_{m-2} & 0 & 0 & 0 & \cdots \\
e_{m-2} & d_{m-1}-k_s & e_{m-1} & 0 & 0 & \cdots \\
0 & e_{m-1} & d_m - k_s & 0 & 0 & \cdots \\
0 & 0 & 0 & d_{m+1} & e_{m+1} & \cdots \\
0 & 0 & 0 & e_{m+1} & d_{m+2} & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \ddots
\end{array}$$

We choose our $\cos\varphi$ and $\sin\varphi$ to be

$$\sin\varphi = s = \frac{e_{m-1}}{\sqrt{e_{m-1}^2 + (d_m - k_s)^2}} \qquad (3.59)$$

$$\cos\varphi = c = \frac{d_m - k_s}{\sqrt{e_{m-1}^2 + (d_m - k_s)^2}} \qquad (3.60)$$

It is easy to see that their squares add up to 1, so they are good cos and sin.

But what if $\sqrt{e_{m-1}^2 + (d_m - k_s)^2} = 0$? This can happen only if $e_{m-1} = 0$ and $d_m - k_s = 0$, or so close to zero that the machine can't make a difference. A condition like that should not really happen on the first sweep, because we have already located such m that $e_m < \varepsilon$, and that m wasn't $m-1$!

This part of the computation is implemented thusly:

```
f = s*e(i)
b = c*e(i)
r = pythag(f, g)
e(i+1) = r
IF(r == 0.0) THEN
   d(i+1) = d(i+1) - p
   e(m) = 0.0
   CYCLE iterate
END IF
s = f/r
c = g/r
```

where s and c stand for sin and cos accordingly.

What are $\sin\varphi$ and $\cos\varphi$ chosen thusly going to do to e_{m-1}?

Remember that we are now looking for a Jacobi rotation that kills element $A_{m-1,m}$ when applied to matrix $\boldsymbol{A}$ from the left, i.e., $\boldsymbol{P}^T \cdot \boldsymbol{A}$:

$$\begin{pmatrix} c & -s \\ s & c \end{pmatrix} \cdot \begin{pmatrix} d_{m-1} - k_s & e_{m-1} \\ e_{m-1} & d_m - k_s \end{pmatrix} = \begin{pmatrix} ? & -ce_{m-1} + s\,(d_m - k_s) \\ ? & ? \end{pmatrix}$$

You can now see that choosing $\sin\varphi$ and $\cos\varphi$ as given by equations (3.59) and (3.60) will indeed do the job.

Now, let us have a look at the next part of the code:

```
g = d(i + 1) - p
r = (d(i) - g) * s + 2.0_sp * c * b
p = s * r
d(i + 1) = g + p
g = c * r - b
```

To understand what happens here we *reverse engineer* the computation from the end, i.e., from the place where

```
d(i + 1) = g + p
```

where `p = s * r`, and $i = m - 1$, in other words, this is:

$$d'_m = g + sr$$

Looking 3 lines up we see that $g = d_m - p$, but initially p is simply 0, so

$$d'_m = d_m + sr$$

In turn

$$r = (d_{m-1} - d_m)\, s + 2ce_{m-1}$$

so

$$d'_m = d_m + s^2 (d_{m-1} - d_m) + 2sce_{m-1} = s^2 d_{m-1} + c^2 d_m + 2sce_{m-1}$$

Now, if you compare this with equation (3.15) on page 54 in section about Jacobi rotations (section 3.2), you will see that this is simply:

$$a'_{qq} = s^2 a_{pp} + c^2 a_{qq} + 2sca_{pq}$$

i.e., a plain Jacobi rotation. This is then followed with the corresponding generation of eigenvectors:

```
IF(PRESENT(z)) THEN
   ff(1:n) = z(1:n,i+1)
   z(1:n, i+1) = s*z(1:n,i) + c*ff(1:n)
   z(1:n,i) = c*z(1:n, i) - s*ff(1:n)
END IF
```

which mimics the following Jacobi formulae:

$$\begin{aligned} v'_{rq} &= sv_{rp} + cv_{rq} \\ v'_{rp} &= cv_{rp} - sv_{rq} \end{aligned}$$

But even though for $i = m - 1$ the operations correspond indeed to a Jacobi rotation, on the following iterations for $i = m-2, m-3, \ldots, l+1, l$ the operations are somewhat different. They are the Givens rotations, whose purpose is to restore the tridiagonal form.

At the end of this process, as we go all the way back to l, d_l and e_l themselves will become affected and e_m will become annihilated, whereas the matrix will remain tridiagonal. This part of the code closes the `iterate` loop:

```
d(l) = d(l) - p
e(l) = g
e(m) = 0.0
```

The innermost DO loop corresponds to subjecting an ever shrinking submatrix of $\boldsymbol{A}$ to a series of QL iterations. The iterations stop once the element e_l has been annihilated.

3.4 Diagonalization of Hermitian Matrices

The problem of diagonalization of Hermitian Matrices reduces trivially to the already solved problem of diagonalization of real symmetric matrices.

Consider the following equation:

$$(\boldsymbol{A} + i\boldsymbol{B}) \cdot (\boldsymbol{u} + i\boldsymbol{v}) = \lambda (\boldsymbol{u} + i\boldsymbol{v})$$

where $\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{u}$ and $\boldsymbol{v}$ are all real and $\boldsymbol{A} = \boldsymbol{A}^T$ and $\boldsymbol{B} = -\boldsymbol{B}^T$, so that $\boldsymbol{C} = \boldsymbol{A} + i\boldsymbol{B}$ is Hermitian. This can be done to any Hermitian matrix.

Separating real and imaginary parts the equation above reduces to:

$$\begin{pmatrix} \boldsymbol{A} & -\boldsymbol{B} \\ \boldsymbol{B} & \boldsymbol{A} \end{pmatrix} \cdot \begin{pmatrix} \boldsymbol{u} \\ \boldsymbol{v} \end{pmatrix} = \lambda \begin{pmatrix} \boldsymbol{u} \\ \boldsymbol{v} \end{pmatrix}$$

The matrix on the left is clearly symmetric because $\boldsymbol{B} = -\boldsymbol{B}^T$.

Observe that if $\begin{pmatrix} \boldsymbol{u} \\ \boldsymbol{v} \end{pmatrix}$ is an eigenvector, then $\begin{pmatrix} -\boldsymbol{v} \\ \boldsymbol{u} \end{pmatrix}$ is also an eigenvector with the same λ_i. This means that the real matrix that corresponds to a Hermitian $\boldsymbol{C}$ has double the number of eigenvalues and they are all degenerate with multiplicity of 2, i.e., λ_1, λ_1, λ_2, λ_2, and so on, and the eigenvectors of $\boldsymbol{C}$ are $\boldsymbol{u} + i\boldsymbol{v}$ and then $i(\boldsymbol{u} + i\boldsymbol{v})$.

Chapter 4

Pushing Particles

4.1 Kinematics and Dynamics of a Material Point

A classical particle or, as it is also referred to, a material point, is an abstraction that does not have a genuine counterpart in nature other than as an idealisation that can be mapped on certain physical systems in some circumstances.

Planets, for example, are so far away from each other and from the Sun that their movements can be described fairly accurately by treating them as material points. But a planet-moon system can no longer be analysed in these terms, because the bodies are sufficiently close to "see" each other's angular dimension and that has a profound effect on the dynamics of a planet-moon system.

All physical bodies, which we encounter in our every-day life have some physical extent, and thus the description of their motion and their interaction with other bodies becomes a very complex issue.

The use of the word "particles" as applied to Quantum particles is a misnomer, that tends to evoke quite inappropriate Classical Physics connotations. Quantum particles are very strange objects. Although they tend to interact with macroscopic apparatuses in a point-like manner, which is why the term "particle" has been slapped onto them, more sophisticated experiments reveal their extended, highly non-local nature. In one of those experiments a quantum particle has been demonstrated to "stretch" over a distance of 15 km and seemingly pass some sort of a communication (not readily accessible to us though) within that stretch at an infinite speed. Quantum Mechanics, as it has been formulated so far, suggests that a quantum particle may even "stretch" like that over the entire universe, which really forces us to revise the very notions of space and time, notions that are intrinsically classical and therefore essentially incompatible with the realm of Quantum Mechanics. Yet we continue to cling to those notions even in Quantum Field Theory, because we have nothing better to replace them with.

The clash between the concepts of a classical space-time continuum and Quantum reality is ultimately responsible for most mathematical difficulties in

Quantum Field Theory. But to discuss those interesting issues in more depth would take us too far from the topic of this chapter, which is numerical simulation of a material point.

4.1.1 Newton's Second Law

The movement of a classical material point is described by the second law of Newton:

$$m\frac{\mathrm{d}^2\boldsymbol{r}(t)}{\mathrm{d}t^2} = \boldsymbol{F}(\boldsymbol{r}, t) \tag{4.1}$$

where $\boldsymbol{r}$ is a vector indicating a position of the material point in space:

$$\boldsymbol{r} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \tag{4.2}$$

and t is time. The representation of $\boldsymbol{r}$ in terms of x, y, and z assumes a Cartesian system of coordinates. In general, i.e., in non-cartesian systems of coordinates, equation (4.1) applies still, but the derivative $\mathrm{d}^2\boldsymbol{r}(t)/\mathrm{d}t^2$ may have to be specially evaluated so that the changes in the directions of the base vectors, as particle moves from A to B are taken into account. This adds the so called connection symbols to the equations.

Vector $\boldsymbol{F}(\boldsymbol{r}, t)$ represents a force field. This force field may be calculated by taking into account interactions with other particles, or interactions with electromagnetic waves, or gravitational fields.

The second law of Newton is an idealisation, of course, even if one was to neglect quantum and relativistic effects. There is no reason why only a second time derivative of $\boldsymbol{r}$ should appear in that equation. Indeed if energy is dissipated in the system usually first time derivatives will appear in the equation too. If a material point loses energy due to electromagnetic radiation, third time derivatives will pop up.

But let us assume, for a moment, that our material point moves in a static force field that can be described in terms of a gradient of some function V:

$$\boldsymbol{F}(\boldsymbol{r}, t) = -q\boldsymbol{\nabla}V(\boldsymbol{r}), \tag{4.3}$$

where q is a *coupling* constant that determines how the material point couples to the force field. Then our Newton equation becomes:

$$m\frac{\mathrm{d}^2\boldsymbol{r}(t)}{\mathrm{d}t^2} = -q\boldsymbol{\nabla}V(\boldsymbol{r}) \tag{4.4}$$

4.1.2 Conservation of Energy and Angular Momentum

Observe that in this system the total energy of the material point, which is defined by:

$$E = \frac{m\dot{\boldsymbol{r}}^2}{2} + qV, \tag{4.5}$$

where $\dot{\boldsymbol{r}} = \mathrm{d}\boldsymbol{r}/\mathrm{d}t$, is conserved. This is easy to see:

$$\begin{aligned}\frac{\mathrm{d}E}{\mathrm{d}t} &= \frac{m}{2}2\dot{\boldsymbol{r}} \cdot \ddot{\boldsymbol{r}} + q\left(\boldsymbol{\nabla} V\right) \cdot \dot{\boldsymbol{r}} \\ &= \dot{\boldsymbol{r}} \cdot (-q\boldsymbol{\nabla} V) + \dot{\boldsymbol{r}} \cdot (q\boldsymbol{\nabla} V) \\ &= 0\end{aligned}$$

If $V(\boldsymbol{r}) = V(r)$, where r is the length of $\boldsymbol{r}$, then another quantity that is also conserved is the angular momentum $\boldsymbol{L} = \boldsymbol{r} \times m\dot{\boldsymbol{r}}$. This is also easy to see:

$$\begin{aligned}\frac{\mathrm{d}\boldsymbol{L}}{\mathrm{d}t} &= \frac{\mathrm{d}}{\mathrm{d}t}\left(\boldsymbol{r} \times m\dot{\boldsymbol{r}}\right) \\ &= \dot{\boldsymbol{r}} \times m\dot{\boldsymbol{r}} + \boldsymbol{r} \times m\ddot{\boldsymbol{r}} \\ &= \boldsymbol{r} \times (-q\boldsymbol{\nabla} V(r))\end{aligned}$$

Now recall that since $V(\boldsymbol{r}) = V(r)$

$$\begin{aligned}\boldsymbol{\nabla} V(r) &= \frac{\mathrm{d}V(r)}{\mathrm{d}r}\boldsymbol{\nabla} r \\ &= \frac{\mathrm{d}V(r)}{\mathrm{d}r}\frac{\boldsymbol{r}}{r}\end{aligned}$$

hence

$$\frac{\mathrm{d}\boldsymbol{L}}{\mathrm{d}t} = \boldsymbol{r} \times \left(-q\frac{\mathrm{d}V(r)}{\mathrm{d}r}\frac{\boldsymbol{r}}{r}\right) = 0$$

4.1.3 Lagrange Equations

Newton equation (4.1), which is a second order differential equation, can be easily reduced to two first order differential equations:

$$\begin{aligned}\frac{\mathrm{d}\boldsymbol{r}}{\mathrm{d}t} &= \boldsymbol{v} \\ \frac{\mathrm{d}\boldsymbol{v}}{\mathrm{d}t} &= -\frac{q}{m}\boldsymbol{\nabla} V\end{aligned}$$

There are various interesting ways, in which these equations can be rewritten. Consider the following scalar function

$$L = \frac{mv^2}{2} - qV \tag{4.6}$$

Note: avoid a confusion with the angular momentum $\boldsymbol{L}$. Traditionally capital "L" is used for both, but function L defined by equation (4.6) is a different thing altogether. It is called a *Lagrangian* and is a function of 6 variables, x, y, z, v_x, v_y, and v_z. Using this function equations (4.6) can be rewritten in the following form:

$$\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial L}{\partial v_x} - \frac{\partial L}{\partial x} &= 0 \\
\frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial L}{\partial v_y} - \frac{\partial L}{\partial y} &= 0 \\
\frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial L}{\partial v_z} - \frac{\partial L}{\partial z} &= 0
\end{aligned} \tag{4.7}$$

It is easy to see that this is indeed the case:

$$\frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial L}{\partial v_x} = \frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial}{\partial v_x}\frac{m\left(v_x^2+v_y^2+v_z^2\right)}{2} = \frac{\mathrm{d}}{\mathrm{d}t}m2\frac{v_x}{2} = m\frac{\mathrm{d}}{\mathrm{d}t}v_x$$

So the first term of the Lagrange equation, $\frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial L}{\partial v_x}$ is simply ma_x. The second term evaluates to:

$$-\frac{\partial L}{\partial x} = -\frac{\partial}{\partial x}\left(-qV\right) = q\nabla_x V$$

Consequently the equation reduces to:

$$ma_x + q\nabla_x V = 0$$

or

$$ma_x = -q\nabla_x V$$

and similarly for y and z components.

An amazing thing about these equations is not that they are equivalent to the Newton equation of motion, but that they look the same also in non-Cartesian coordinates. Mathematicians and physicists often write them in the following form:

$$\frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial L}{\partial \dot{q}_i} - \frac{\partial L}{\partial q_i} = 0, \quad i = 1, 2, 3 \tag{4.8}$$

where q_i is referred to as a *generalized* coordinate, and a dot above a symbol, as before, denotes a time derivative.

4.1.4 Hamilton Equations

Another way in which Newton equations are often rewritten is due to Hamilton:

$$\begin{aligned}
\dot{x} &= \frac{\partial E}{\partial p_x} \\
\dot{p_x} &= -\frac{\partial E}{\partial x} \\
\dot{y} &= \frac{\partial E}{\partial p_y} \\
\dot{p_y} &= -\frac{\partial E}{\partial y}
\end{aligned}$$

$$\begin{aligned}\dot{z} &= \frac{\partial E}{\partial p_z} \\ \dot{p_z} &= -\frac{\partial E}{\partial z}\end{aligned} \tag{4.9}$$

where $\boldsymbol{p} = m\boldsymbol{v}$ is the momentum of a material point.

It is easy to see that these equations are equivalent to the Newton's equation too. In order to see that first observe that

$$\frac{mv^2}{2} = \frac{p^2}{2m}$$

Hence the total energy of a material point can be rewritten as:

$$E = \frac{p^2}{2m} + qV$$

Now you can easily see that:

$$\frac{\partial E}{\partial p_x} = \frac{\partial}{\partial p_x}\frac{p_x^2 + p_y^2 + p_z^2}{2m} = \frac{p_x}{m} = \frac{mv_x}{m} = v_x = \dot{x}$$

so the equation

$$\dot{x} = \frac{\partial E}{\partial p_x}$$

simply states that $\dot{x} = v_x = p_x/m$. The second Hamilton equation states that:

$$\dot{p}_x = m\dot{v}_x = ma_x = -\frac{\partial qV}{\partial x}$$

so it is equivalent to the Newton equation.

Hamilton equations are often expressed in terms of generalized coordinates too and written in the following way:

$$\begin{aligned}\dot{q}_i &= \frac{\partial H}{\partial p_i} \\ \dot{p}_i &= -\frac{\partial H}{\partial q_i}, \quad i = 1, 2, 3\end{aligned} \tag{4.10}$$

where H is the Hamilton function, or the *Hamiltonian*. It is, as you have seen, simply a total energy of the material point.

4.1.5 Lagrange and Hamilton Equations for Many-Particle Systems

Lagrange equations (4.8) and Hamilton equations (4.10) are valid also for an ensemble of material points interacting with each other and with some external potentials. In this case

$$L\left(q_1,\ldots,q_n,\dot{q}_1,\ldots,\dot{q}_n\right)=\sum_{i=1}^{n}\frac{m_i\dot{q}_i^2}{2}-U(q_1,q_2,\ldots,q_n), \tag{4.11}$$

where $U(q_1,q_2,\ldots,q_n)$ is the potential energy of the multi-particle system, and

$$H\left(q_1,\ldots,q_n,p_1,\ldots,p_n\right)=\sum_{i=1}^{n}p_iq_i-L \tag{4.12}$$

Observe how these two functions are defined in terms of coordinates q_i and either velocities $\dot{q}_i$ or momenta p_i. Hamiltonian is a function defined on the *phase space* of a system.

4.1.6 Poisson Brackets

For functions that are defined on the phase space we can define the following operation. Let $F=F(\boldsymbol{q},\boldsymbol{p},t)$ and $G=G(\boldsymbol{q},\boldsymbol{p},t)$. Then a *Poisson* bracket of these two functions is defined by:

$$\{F,G\}=\sum_{i=1}^{n}\left(\frac{\partial F}{\partial q_i}\frac{\partial G}{\partial p_i}-\frac{\partial G}{\partial q_i}\frac{\partial F}{\partial p_i}\right) \tag{4.13}$$

This operation has the following neat properties:

$$\begin{aligned}
\{F,G\} &= -\{G,F\} && (4.14)\\
\{F,F\} &= 0 && (4.15)\\
\{F_1+F_2,G\} &= \{F_1,G\}+\{F_2,G\} && (4.16)\\
\{F_1F_2,G\} &= F_1\{F_2,G\}+F_2\{F_1,G\} && (4.17)\\
\{F,q_i\} &= -\frac{\partial F}{\partial p_i} && (4.18)\\
\{F,q_i\} &= \frac{\partial F}{\partial q_i} && (4.19)\\
\{q_i,q_j\} &= 0 && (4.20)\\
\{p_i,p_j\} &= 0 && (4.21)\\
\{q_i,p_j\} &= \delta_{ij} && (4.22)\\
0 &= \{F_1,\{F_2,F_3\}\}+\{F_2,\{F_3,F_1\}\}+\{F_3,\{F_1,F_2\}\} && (4.23)\\
\frac{\partial}{\partial t}\{F,G\} &= \left\{\frac{\partial F}{\partial t},G\right\}+\left\{F,\frac{\partial G}{\partial t}\right\} && (4.24)\\
& && (4.25)
\end{aligned}$$

Poisson brackets can be used to express time derivatives of phase space functions:

$$\frac{\mathrm{d}F}{\mathrm{d}t} = \sum_{i=1}^{n}\left(\frac{\partial F}{\partial q_i}\frac{\mathrm{d}q_i}{\mathrm{d}t}+\frac{\partial F}{\partial p_i}\frac{\mathrm{d}p_i}{\mathrm{d}t}\right)+\frac{\partial F}{\partial t}$$

$$\begin{aligned} &= \sum_{i=1}^{n}\left(\frac{\partial F}{\partial q_i}\frac{\partial H}{\partial p_i} - \frac{\partial F}{\partial p_i}\frac{\partial H}{\partial q_i}\right) + \frac{\partial F}{\partial t} \\ &= \{F, H\} + \frac{\partial F}{\partial t} \qquad (4.26) \end{aligned}$$

This equation can then be applied to q_i and p_i itself to re-express the Hamilton equations in the following form:

$$\frac{\mathrm{d}q_i}{\mathrm{d}t} = \{q_i, H\} \qquad (4.27)$$

$$\frac{\mathrm{d}p_i}{\mathrm{d}t} = \{p_i, H\} \qquad (4.28)$$

In turn, substituting H in place of F yields:

$$\frac{\mathrm{d}H}{\mathrm{d}t} = \{H, H\} + \frac{\partial H}{\partial t} = \frac{\partial H}{\partial t} \qquad (4.29)$$

Expressions such as $\{q_i, p_j\} = \delta_{ij}$ ought to tug at the heart of everyone acquainted with Quantum Mechanics, where one of the expressions of the Heisenberg Uncertainty Principle is

$$[\hat{q}_i, \hat{p}_j] = i\hbar\delta_{ij},$$

where

$$[\hat{q}_i, \hat{p}_j] = \hat{q}_i\hat{p}_j - \hat{p}_j\hat{q}_i$$

is a *commutator* of operators that represent position and momentum. Similarly time evolution of any Quantum Mechanical operator that does not depend on time explicitly is given by

$$\left[\hat{\Psi}, \hat{H}\right] = i\hbar\frac{\mathrm{d}\hat{\Psi}}{\mathrm{d}t}$$

This is not entirely an accident. Poisson brackets lead directly to the so called canonical quantization. Canonical quantization is a procedure which converts a classical field theory or a classical mechanical theory into the corresponding Quantum theory. One of its rules is:

$$\{\Psi, \Phi\} \rightarrow \frac{1}{i\hbar}\left[\hat{\Psi}, \hat{\Phi}\right]$$

But the truth about canonical quantization carried out like that is that it has to be interfered with frequently in order to deliver a meaningful Quantum theory, and the reason for that is that Quantum theories cannot be derived formally from classical theories. The opposite is the case, i.e., Quantum theories are a lot richer than classical theories, and it is the latter that are derivable from the former in thermodynamic limit. But canonical quantization was useful in its day in providing a bridge between XIXth century classical physics and XXth century quantum physics.

4.1.7 Hamilton-Jacobi Equation

There is nothing stopping us from transforming variables on the phase space from $(q_i, p_i), i = 1, 2, \ldots, n$ to some new $(Q_i, P_i), i = 1, 2, \ldots, n$. Normally, if you change variables this way the functional form of a Hamiltonian is going to change too. In general it is not the case that if (q_i, p_i) and H satisfy Hamilton equations then the new (Q_i, P_i) and $\bar{H}$ will satisfy Hamilton equations too.

But it turns out that if the transformation $(q_i, p_i) \to (Q_i, P_i)$ satisfies certain conditions then Hamilton equations are preserved. Such transformations are called *canonical transformations* and the condition, it turns out, is as follows: for a transformation to be *canonical* there must exist a function

$$S(q_1, \ldots, q_n, P_1, \ldots, P_n, t),$$

called a *forming* or a *generating* function, such that

$$p_i = \frac{\partial S(\boldsymbol{q}, \boldsymbol{P}, t)}{\partial q_i} \tag{4.30}$$

$$Q_i = \frac{\partial S(\boldsymbol{q}, \boldsymbol{P}, t)}{\partial P_i} \tag{4.31}$$

$$\bar{H}(\boldsymbol{Q}, \boldsymbol{P}, t) = H(\boldsymbol{q}, \boldsymbol{p}, t) + \frac{\partial S}{\partial t} \tag{4.32}$$

This observation provides us with the means to solve Hamilton equations and find quite easily all constants of motion. Imagine that we have found a canonical transformation, given by some forming function S, such that the new Hamiltonian is zero. Then from Hamilton equations it follows that:

$$\begin{aligned} \dot{Q}_i &= \frac{\partial \bar{H}}{\partial P_i} = 0 \\ \dot{P}_i &= -\frac{\partial \bar{H}}{\partial Q_i} = 0 \end{aligned}$$

therefore Q_i and P_i must be constants of motion and function $S(\boldsymbol{q}, \boldsymbol{P}, t) = {}^{\boldsymbol{P}}S(\boldsymbol{q}, t)$ is parametrised by $\boldsymbol{P}$ only, and, furthermore,

$$\begin{aligned} q_i &= q_i(\boldsymbol{Q}, \boldsymbol{P}, t) = {}^{\boldsymbol{Q},\boldsymbol{P}}q_i(t) \\ p_i &= p_i(\boldsymbol{Q}, \boldsymbol{P}, t) = {}^{\boldsymbol{Q},\boldsymbol{P}}p_i(t) \end{aligned}$$

If S is to be a forming function of a canonical transformation then we must have *first* that:

$$p_i = \frac{\partial S(\boldsymbol{q}, \boldsymbol{P}, t)}{\partial q_i}$$

and *then* substituting this condition into $H(\boldsymbol{q}, \boldsymbol{p}, t)$, we obtain:

$$H\left(q_1, \ldots, q_n, \frac{\partial S}{\partial q_1}, \ldots, \frac{\partial S}{\partial q_n}, t\right) + \frac{\partial S}{\partial t} = 0 \tag{4.33}$$

This is the Hamilton-Jacobi equation.

The Hamilton-Jacobi equation is enormously useful in solving analytically and numerically equations of motion for classical particles. The main reason for its usefulness is that it yields all constants of motion automatically, and the solution itself becomes formulated in terms of those constants of motion.

Another interesting feature of this equation is that the forming function S behaves a little like a wave. It can be shown that particle trajectories pierce surfaces of constant S.

A yet another interesting feature of the Hamilton-Jacobi equation is that it can be easily derived from the Schrödinger equation of Quantum Mechanics by representing the wave function $\Psi(\boldsymbol{r}, t)$ in the polar form

$$\Psi(\boldsymbol{r}, t) = \mathcal{A}(\boldsymbol{r}, t) e^{iS(\boldsymbol{r},t)/\hbar} \tag{4.34}$$

where $\mathcal{A}$ and S are both real.

Here is how this comes about.

Start from the Schrödinger equation for a single quantum "particle":

$$i\hbar \frac{\partial \Psi}{\partial t} = -\frac{\hbar^2}{2m} \nabla^2 \Psi + U\Psi, \tag{4.35}$$

where $\hbar = h/(2\pi)$, and h is the Planck constant, Ψ is the *wave function* of the particle, and U is a potential. For example we can have $U(\boldsymbol{r}) = qV(\boldsymbol{r})$, where q is an electric charge of the particle. Now substitute equation (4.34) in place of Ψ. This yields for $\partial\Psi/\partial t$:

$$\frac{\partial \Psi}{\partial t} = e^{iS/\hbar} \left(\frac{\partial \mathcal{A}}{\partial t} + \frac{i}{\hbar} \mathcal{A} \frac{\partial S}{\partial t} \right) \tag{4.36}$$

There is a little more work that we have to do with the Laplacian on the right hand side of the Schrödinger equation. Let us evaluate $\boldsymbol{\nabla}\Psi$ first:

$$\boldsymbol{\nabla}\Psi = e^{iS/\hbar} \left(\boldsymbol{\nabla}\mathcal{A} + \frac{i}{\hbar} \mathcal{A} \boldsymbol{\nabla} S \right) \tag{4.37}$$

Acting with $\boldsymbol{\nabla}$ again yields:

$$\nabla^2 \Psi = e^{iS/\hbar} \left(\nabla^2 \mathcal{A} - \frac{\mathcal{A}}{\hbar^2} (\boldsymbol{\nabla} S)^2 + \frac{i}{\hbar} \left(2\boldsymbol{\nabla}\mathcal{A} \cdot \boldsymbol{\nabla} S + \mathcal{A} \nabla^2 S \right) \right) \tag{4.38}$$

Now we have to substitute equations (4.36) and (4.38) into the Schrödinger equation, divide both sides by $e^{iS/\hbar}$, this term will accompany all other terms, and collect separately real and imaginary parts of the resulting equation.

Collecting the real part yields:

$$-\mathcal{A} \frac{\partial S}{\partial t} = -\frac{\hbar^2}{2m} \left(\nabla^2 \mathcal{A} - \frac{\mathcal{A}}{\hbar^2} (\boldsymbol{\nabla} S)^2 \right) + U\mathcal{A} \tag{4.39}$$

Dividing this equation by $-\mathcal{A}$ and grouping all terms where $\hbar$ cancels out on the left hand side yields:

$$\frac{\partial S}{\partial t} + \frac{(\boldsymbol{\nabla} S)^2}{2m} + U = \frac{\hbar^2}{2m}\frac{\nabla^2 \mathcal{A}}{\mathcal{A}} \tag{4.40}$$

The imaginary part, in turn, yields

$$\hbar \frac{\partial A}{\partial t} = -\frac{\hbar^2}{2m}\frac{1}{\hbar}\left(2\boldsymbol{\nabla}\mathcal{A}\cdot\boldsymbol{\nabla}S + \mathcal{A}\nabla^2 S\right) \tag{4.41}$$

Multiplying both sides of this equation by $2\mathcal{A}/\hbar$ lets us rewrite it finally as:

$$\frac{\partial \mathcal{A}^2}{\partial t} = -\boldsymbol{\nabla}\cdot\left(\mathcal{A}^2\frac{\boldsymbol{\nabla}S}{m}\right) \tag{4.42}$$

If you look at the real part of the Schrödinger equation, equation (4.40), the one that gives $\partial S/\partial t$, you can see that in the limit $\hbar \to 0$ it turns into:

$$\frac{\partial S}{\partial t} + \frac{(\boldsymbol{\nabla} S)^2}{2m} + U = 0 \tag{4.43}$$

which is the Hamilton-Jacobi equation for a classical particle moving in a potential field U. The neglected term proportional to Planck constant

$$\frac{\hbar^2}{2m}\frac{\nabla^2 \mathcal{A}}{\mathcal{A}} \doteq -Q \tag{4.44}$$

is called a *quantum potential.* For wave functions that are valid solutions of the Schrödinger equation, this quantum potential does not vanish as we move away from a quantum "particle". It is the source of non-locality of Quantum Mechanics. Using this non-vanishing quantum potential particles can feel their way around as they travel through space. Or, since the quantum potential is actually made of the particle's wave function, we can just as well say, that a quantum particle *spreads* as far as its quantum potential, the potential itself being an integral component of what a quantum particle is.

But returning to the Hamilton-Jacobi equation, in summary, what we have just demonstrated is that:

> *The Hamilton-Jacobi function S is, on the one hand, the forming function of a canonical transformation that annihilates the Hamiltonian, and, on the other, it is the* phase *of the quantum mechanical wave function that represents a quantum particle.*

4.1.8 Solving the Hamilton-Jacobi Equation

In this section we're going to use the Hamilton-Jacobi equation in order to derive analytical formulae for a motion of a material point in the central Newtonian M/r potential.

The derived formulae can then be used to compare an approximate numerical solution against an analytical, i.e., exact solution to the problem.

The problem is easiest to describe in spherical coordinates, r, θ and ϕ. In these coordinates the Hamiltonian assumes the following form:

$$H = \frac{p_r^2}{2} + \frac{p_\theta^2}{2r^2} + \frac{p_\phi^2}{2r^2 \sin^2\theta} - \frac{M}{r} \tag{4.45}$$

where we have assumed that the gravitational constant $G = 1$, and the mass of the material point $m = 1$ too.

The Hamilton-Jacobi equation that corresponds to that Hamiltonian is

$$\frac{1}{2}\left(\frac{\partial S}{\partial r}\right)^2 + \frac{1}{2r^2}\left(\frac{\partial S}{\partial \theta}\right)^2 + \frac{1}{2r^2\sin^2\theta}\left(\frac{\partial S}{\partial \phi}\right)^2 - \frac{M}{r} + \frac{\partial S}{\partial t} = 0 \tag{4.46}$$

The method that is commonly use here is called the *separation of variables*. The Hamiltonian does not depend explicitly on

- time, t
- angle ϕ

Therefore a solution is sought in the following form:

$$S = -Et + p_\phi \phi + S_r(r) + S_\theta(\theta) + C, \tag{4.47}$$

where C is a constant.

Substituting this into the Hamilton-Jacobi equation yields:

$$\frac{1}{2}\left(\frac{\mathrm{d}S_r}{\mathrm{d}r}\right)^2 + \frac{1}{2r^2}\left(\frac{\mathrm{d}S_\theta}{\mathrm{d}\theta}\right)^2 + \frac{p_\phi^2}{2r^2\sin^2\theta} - \frac{M}{r} - E = 0 \tag{4.48}$$

Multiply this equation by $2r^2$. Now we can rewrite this equation placing all terms that depend on θ on the left hand side and all terms that depend on r on the right hand side:

$$\left(\frac{\mathrm{d}S_\theta}{\mathrm{d}\theta}\right)^2 + \frac{p_\phi^2}{\sin^2\theta} = 2Mr + 2Er^2 - r^2\left(\frac{\mathrm{d}S_r}{\mathrm{d}r}\right)^2 \tag{4.49}$$

Because expressions on both sides of this equation depend on different variables, the equality can hold only if they are equal to the same constant, L^2:

$$\left(\frac{\mathrm{d}S_\theta}{\mathrm{d}\theta}\right)^2 + \frac{p_\phi^2}{\sin^2\theta} = L^2 = 2Mr + 2Er^2 - r^2\left(\frac{\mathrm{d}S_r}{\mathrm{d}r}\right)^2 \tag{4.50}$$

And this implies that:

$$\frac{\mathrm{d}S_r}{\mathrm{d}r} = \sqrt{2\left(\frac{M}{r} + E - \frac{L^2}{2r^2}\right)} \tag{4.51}$$

$$\frac{\mathrm{d}S_\theta}{\mathrm{d}\theta} = \sqrt{L^2 - \frac{p_\phi^2}{\sin^2\theta}} \tag{4.52}$$

These, in turn, are first order ordinary differential equations, which can be readily integrated.

But recall that there are additional conditions that function S must satisfy. In this case

$$S = S(r, \theta, \phi, P_r, P_\theta, P_\phi, t) \tag{4.53}$$

where P_r, P_θ and P_ϕ are constants and

$$\frac{\partial S}{\partial P_r} = Q_r \tag{4.54}$$

$$\frac{\partial S}{\partial P_\theta} = Q_\theta \tag{4.55}$$

$$\frac{\partial S}{\partial P_\phi} = Q_\phi \tag{4.56}$$

where Q_r, Q_ϕ and Q_θ are also constants. In our case we have $P_r \sim E$, $P_\phi \sim p_\phi$, and $P_\theta \sim L$, and we can set Q_r, Q_ϕ and Q_θ to zero so that:

$$\frac{\partial S}{\partial E} = 0 \tag{4.57}$$

$$\frac{\partial S}{\partial L} = 0 \tag{4.58}$$

$$\frac{\partial S}{\partial p_\phi} = 0 \tag{4.59}$$

The last equation (4.59) yields:

$$0 = \frac{\partial S}{\partial p_\phi} = \phi - \int_{\theta_0}^{\theta} \frac{(p_\phi/L)\,\mathrm{d}\theta}{\sin\theta\sqrt{\sin^2\theta - p_\phi^2/L^2}} \tag{4.60}$$

It can be proven that this relation is satisfied by a flat motion, i.e., that the material point moves in a plane with vector $\boldsymbol{L}$ perpendicular to that plane. We can therefore change our system of coordinates so that, say, $p_\phi = 0$. Then

$$S = -Et + L\theta + \int_{r_0}^{r_1} \sqrt{2\left(E + \frac{M}{r} - \frac{L^2}{2r^2}\right)} + C \tag{4.61}$$

Equation (4.58) yields the shape of the orbit:

$$0 = \theta - \int_{r_0}^{r_1} \frac{L\,\mathrm{d}r/r^2}{\sqrt{2\left(E + \frac{M}{r} - \frac{L^2}{2r}\right)}} \tag{4.62}$$

with the following solution:

$$r = \frac{L^2/M}{1 + e\cos\theta} \tag{4.63}$$

where e is the *eccentricity* of the orbit:

$$e = \sqrt{1 + \frac{2EL^2}{M^2}} \tag{4.64}$$

Other parameters pertaining to the orbit are:

$$a = -\frac{M}{2E} \tag{4.65}$$

Remember that for a trapped particle the energy is negative. a is the semimajor axis of the orbit (when elliptic).

$$b = \frac{L}{\sqrt{-2E}} \tag{4.66}$$

b is the semiminor axis of the orbit (when elliptic)

$$r_{\text{min}} = \frac{L^2/M}{1 + \sqrt{1 + \frac{2EL^2}{M^2}}} \tag{4.67}$$

r_{min} is the distance of the closest approach.

Equation (4.57) yields the time dependence of r versus t:

$$0 = -t + \int_{r_0}^{r_1} \frac{\mathrm{d}r}{\sqrt{2\left(E + \frac{M}{r} - \frac{L^2}{2r^2}\right)}} \tag{4.68}$$

The solution to this equation is quite complicated and can be given in terms of Bessel functions and harmonic motion with the mean circular frequency of

$$\frac{2\pi}{T} = \omega = \sqrt{\frac{M}{a^3}} \tag{4.69}$$

4.2 Euler Method

Since Newton equations can be reduced either directly or through the Lagrangian or Hamiltonian routes to systems of first order ordinary differential equations, once we know how to solve the latter numerically, we'll have a handle on Newton equations themselves.

So let us consider then the following simple ODE:

$$\frac{\mathrm{d}x(t)}{\mathrm{d}t} = f(x,t) \tag{4.70}$$

The Euler method of solving this equation is very simple. Replace $\mathrm{d}x(t)/\mathrm{d}t$ with a finite difference:

$$\frac{x(t + \Delta t) - x(t)}{\Delta t} = f\left(x(t), t\right) \tag{4.71}$$

This is, actually, *one* possible choice, that will lead to an *explicit* integration scheme. The other choice is:

$$\frac{x(t+\Delta t)-x(t)}{\Delta t} = f\left(x(t+\Delta t), t+\Delta t\right) \tag{4.72}$$

This choice leads to an *implicit* integration scheme.

Let us focus on the explicit scheme first. Equation (4.71) suggests the following integration scheme:

$$x(t+\Delta t) = x(t) + f\left(x(t), t\right)\Delta t \tag{4.73}$$

or, in other words, once we have assumed a certain initial condition for x, say, $x_0 = x(t_0)$ then:

$$\begin{aligned} x_1 &= x(t_1) = x(t_0+\Delta t) = x_0 + f(x_0,t_0)\Delta t \\ x_2 &= x(t_2) = x(t_1+\Delta t) = x_1 + f(x_1,t_1)\Delta t \\ x_3 &= x(t_3) = x(t_2+\Delta t) = x_2 + f(x_2,t_2)\Delta t \\ &\dots \end{aligned}$$

Observe that this scheme is intrinsically sequential. It cannot be parallelised, unless you are solving a very large system of coupled ODEs, in which case, the whole system of equations can be solved in parallel. This is generally true of any ODE problem, and any ODE numerical method. They cannot be parallelised by themselves. Parallelism is possible for large systems of ODEs only.

The Euler scheme may become unstable. To see that consider two possible solutions x_n and x'_n at t_n that are very close, so that $x'_n = x_n + \delta x_n$, where δx_n is a very small number. In order to assess the stability of the method we need to consider the evolution of δx_n with time.

Plugging $x_n + \delta x_n$ into the Euler scheme yields:

$$x_{n+1}+\delta x_{n+1} = x_n+\delta x_n+f\left(x_n+\delta x_n,t_n\right)\Delta t \approx x_n+\delta x_n+f\left(x_n,t_n\right)\Delta t+\left.\frac{\partial f(x,t)}{\partial x}\right|_{\substack{x=x_n\\t=t_n}}\delta x_n\Delta t \tag{4.74}$$

Since x_{n+1} and x_n are already linked by the Euler relationship, we get

$$\delta x_{n+1} = \delta x_n + \left.\frac{\partial f(x,t)}{\partial x}\right|_{\substack{x=x_n\\t=t_n}}\delta x_n\Delta t \tag{4.75}$$

Now, let us assume that $\delta x_{n+1} = g x_n$, where g is a *growth factor*. Plugging this into equation (4.75) yields:

$$g = 1 + \left.\frac{\partial f(x,t)}{\partial x}\right|_{\substack{x=x_n\\t=t_n}}\Delta t \tag{4.76}$$

For the scheme to be numerically stable we must have that $g \in [-1,1]$, hence the condition:

$$\left.\frac{\partial f(x,t)}{\partial x}\right|_{\substack{x=x_n\\t=t_n}}\Delta t \in [-2,0] \tag{4.77}$$

This means that if, for example, $\frac{\partial f(x,t)}{\partial x}\Big|_{\substack{x=x_n\\t=t_n}}$ is positive then Euler method is bound to be unstable, whatever the choice of Δt. If however $\frac{\partial f(x,t)}{\partial x}\Big|_{\substack{x=x_n\\t=t_n}}$ is negative, then by choosing an appropriately small Δt we can stabilise the computation.

The simplest example of a negative $\frac{\partial f(x,t)}{\partial x}\Big|_{\substack{x=x_n\\t=t_n}}$ is when it is a negative constant, e.g.,

$$\frac{\partial f(x,t)}{\partial x}\Bigg|_{\substack{x=x_n\\t=t_n}} = -k, \tag{4.78}$$

where $k > 0$, or, in other words,

$$\frac{\mathrm{d}x(t)}{\mathrm{d}t} = -kx + C \tag{4.79}$$

Assume that $C = 0$, then the exact solution is

$$x(t) = x_0 e^{-kt} \tag{4.80}$$

The stability criterion implies that $\Delta t \in [0, 2/k]$.

Euler method is not only unstable in great many circumstances. It is not very accurate either. This can be ascertained easily by comparing the exact solution of

$$\frac{\mathrm{d}x(t)}{\mathrm{d}t} = -kx$$

for which a stable numerical Euler scheme is possible, with a numerical approximation. Starting from x_0 numerically we are going to get:

$$x_1 = x(\Delta t) = x_0 - kx_0\Delta t$$

The exact solution is:

$$x_1 = x_0 e^{-k\Delta t} = x_0\left(1 - k\Delta t + \frac{(k\Delta t)^2}{2!} - \frac{(k\Delta t)^3}{3!} + \ldots\right)$$

Hence the error is:

$$\delta x_1 = x_0\left(\frac{(k\Delta t)^2}{2!} - \frac{(k\Delta t)^3}{3!} + \ldots\right)$$

or of order $\mathcal{O}\left((\Delta t)^2\right)$. Such methods are said to be *first-order* accurate.

The reason for this lack of accuracy is that

> *as the formula advances the solution through the interval* $[t_n, t_n + \Delta t]$ *the derivative information that pertains to the interval is sampled only at the beginning of the interval.*

4.3 The Mid-Point Method

One of the remedies to the problem that we have with the Euler method is to sample more frequently. Consider, for example, the following scheme:

$$\begin{aligned} k_1 &= \Delta t f\left(x_n, t_n\right) \\ k_2 &= \Delta t f\left(x_n + \frac{1}{2}k_1, t_n + \frac{1}{2}\Delta t\right) \\ x_{n+1} &= x_n + k_2 + \mathcal{O}\left((\Delta t)^3\right) \end{aligned} \tag{4.81}$$

We can combine all those three equations together to obtain the following expression:

$$x_{n+1} = x_n + f\left(x_n + f\left(x_n, t_n\right)\frac{1}{2}\Delta t, t_n + \frac{1}{2}\Delta t\right)\Delta t \tag{4.82}$$

In other words, in this scheme we make a simple Euler step to the mid-point of the interval $[t_n, t_n+\Delta t]$, find the corresponding $x_{n+1/2}$ and evaluate $f\left(x_{n+1/2}, t_{n+1/2}\right)$, and then use *that* value in order to jump from t_n to t_{n+1}.

The mid-point method is second-order accurate.

The mid-point method is the simplest example of the family of Runge-Kutta methods. It is also referred to as the *second-order* Runge-Kutta method.

4.4 The Fourth-Order Runge-Kutta Method

The most often used method of the Runge-Kutta family is the Fourth-Order one, which extends the idea of the mid-point method, by jumping 1/4th of the way first, then going half-way, a la the mid-point method, then going 3/4th of the way and finally juping all the way.

The formula for this method looks as follows:

$$\begin{aligned} k_1 &= f\left(x_n, t_n\right)\Delta t \\ k_2 &= f\left(x_n + k_1/2, t_n + \Delta t/2\right)\Delta t \\ k_3 &= f\left(x_n + k_2/2, t_n + \Delta t/2\right)\Delta t \\ k_4 &= f\left(x_n + k_3, t_n + \Delta t\right)\Delta t \\ x_{n+1} &= x_n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} + \mathcal{O}\left((\Delta t)^5\right) \end{aligned} \tag{4.83}$$

4.4.1 Example Code

Here is a simple example code that implements the Fourth-Order Runge-Kutta:

```
SUBROUTINE rk4(y, dydx, x, h, yout, derivs)

  USE nrtype; USE nrutil, ONLY : assert_eq
  IMPLICIT NONE
  REAL(sp), DIMENSION(:), INTENT(in) :: y, dydx
  REAL(sp), INTENT(in) :: x, h
  REAL(sp), DIMENSION(:), INTENT(out) :: yout

  INTERFACE
     SUBROUTINE derivs(x, y, dydx)
       USE nrtype
       IMPLICIT NONE
       REAL(sp), INTENT(in) :: x
       REAL(sp), DIMENSION(:), INTENT(in) :: y
       REAL(sp), DIMENSION(:), INTENT(out) :: dydx
     END SUBROUTINE derivs
  END INTERFACE

  INTEGER(i4b) :: ndum
  REAL(sp) :: h6, hh, xh
  REAL(sp), DIMENSION(SIZE(y)) :: dym, dyt, yt

  ndum = assert_eq(SIZE(y), SIZE(dydx), SIZE(yout), 'rk4')
  hh = h*0.5_sp
  h6 = h/6.0_sp
  xh = x + hh
  yt = y + hh * dydx
  CALL derivs(xh, yt, dyt)
  yt = y + hh * dyt
  CALL derivs(xh, yt, dym)
  yt = y + h * dym
  dym = dyt + dym
  CALL derivs(x + h, yt, dyt)
  yout = y + h6 * (dydx + dyt + 2.0_sp * dym)

END SUBROUTINE rk4
```

The subroutine applies Runge-Kutta solver to a system of equations:

$$\begin{aligned}
\frac{\mathrm{d}y_1}{\mathrm{d}x} &= \mathrm{dydx}_1\,(y_1, y_2, \ldots, y_m, x) \\
\frac{\mathrm{d}y_2}{\mathrm{d}x} &= \mathrm{dydx}_2\,(y_1, y_2, \ldots, y_m, x) \\
&\cdots \\
\frac{\mathrm{d}y_m}{\mathrm{d}x} &= \mathrm{dydx}_m\,(y_1, y_2, \ldots, y_m, x)
\end{aligned}$$

The computation is carried on in parallel, assuming a parallelising Fortran compiler.

Note that here the lower index numbers the equations not time steps. Furthermore the notation itself is $\boldsymbol{y}(x)$ rather than $\boldsymbol{x}(t)$.

The arguments that must be passed to the subroutine are

y which is the vector of initial values of $\boldsymbol{y}_n$ (in our notation this would be $\boldsymbol{x}_n$) at "time" x_n (in our notation this would be t_n);

dydx which is the vector of what in our notation would be $\boldsymbol{f}(\boldsymbol{x}_n, t_n)$, and in this program's notation, these are the values that the right hand side assumes at "time" x_n;

x which is the value of time, in our notation t_n;

h which is the length of the time step, in our notation Δt;

yout which is a place for the new, updates values of $\boldsymbol{y}(x + h)$; in our notation this would be $\boldsymbol{x}_{n+1} = \boldsymbol{x}(t + \Delta t)$;

derivs which is the name of the function that corresponds to our function $f(\boldsymbol{x}, t)$, and whose job is to evaluate dydx at various stages of the computation.

The first operation simply checks if arrays `y`, `dydx`, and `yout` are of matching sizes, and, if not, an appropriate error message is written on standard output. Otherwise the size is written on a dummy variable `ndum` that is not used any more:

```
ndum = assert_eq(SIZE(y), SIZE(dydx), SIZE(yout), 'rk4')
```

The next three statements:

```
hh = h*0.5_sp
h6 = h/6.0_sp
xh = x + hh
```

evaluate $\frac{1}{2}\Delta t$ (`hh`), $\frac{1}{6}\Delta t$ (`h6`), and $t + \frac{1}{2}\Delta t$, which goes into `xh`.

And now we evaluate

$$x_n + \frac{k_1}{2} = x_n + f(x_n, t_n)\frac{\Delta t}{2},$$

and store it on `yt`. This will go into k_2 later:

```
yt = y + hh * dydx
```

The next step is to evaluate

$$k_2 = f\left(x_n + f(x_n, t_n)\frac{\Delta t}{2}, t_n + \frac{\Delta t}{2}\right)\Delta t,$$

where $x_{n+1/2}$ is already stored on `yt`, and $t_{n+1/2}$ is stored on xh. Observe that subroutine `derivs` reverses the order of arguments, compared with our notation, i.e., time (`xh`) goes first, then position (`yt`), and the last argument is used for the answer, once the subroutine returns:

```
CALL derivs(xh, yt, dyt)
```

What this function actually returns, is not k_2, but $k_2/\Delta t$, because we haven't multiplied `dyt` by anything yet.

This multiplication takes place in the next line, when we evaluate

```
yt = y + hh * dyt
```

But remember that hh is really $\Delta t/2$, so what we are evaluating here, in fact is:

$$x_n + \frac{k_2}{2}$$

so that the next call:

```
CALL derivs(xh, yt, dym)
```

evaluates:

$$f\left(x_n + \frac{k_2}{2}, t_n + \frac{\Delta t}{2}\right),$$

which really is $k_3/\Delta t$, and returns it in dym.

Now we evaluate $x_n + k_3$ thusly:

```
yt = y + h * dym
```

Remember that whereas hh stands for $\Delta t/2$, the single h is just Δt, so `h * dym` is indeed plain k_3, whereas `x + h` stands for, in our notation, $t_n + \Delta t$. With these two we can now evaluate k_4:

```
CALL derivs(x + h, yt, dyt)
```

What's returned in dyt is $k_4/\Delta t$.

Observe that just before calling derivs we have save the previous value of `dyt + dym` on dym. That previous value was $k_2/\Delta t + k_3/\Delta t$. So this means that `h6 * 2.0_sp * dym` stands for

$$\frac{\Delta t}{6} 2\left(\frac{k_2}{\Delta t} + \frac{k_3}{\Delta t}\right) = \frac{k_2}{3} + \frac{k_3}{3}$$

In turn `h6 * (dydx + dyt)`, stands for

$$\frac{\Delta t}{6}\left(\frac{k_1}{\Delta t} + \frac{k_4}{\Delta t}\right) = \frac{k_1}{6} + \frac{k_4}{6}$$

Consequently, you can now see clearly that:

```
yout = y + h6 * (dydx + dyt + 2.0_sp * dym)
```

evaluates

$$x_n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6}$$

which is the Runge-Kutta formula.

4.4.2 Adaptive Stepsize Control

In order to assess the accuracy of numerical integration, and possibly adjust the *stepsize* so as to maintain the requested accuracy employ *step doubling*:

1. take the step twice
   (a) once as a full step, leading to $x_1(t+\Delta t)$
   (b) then as two half steps, leading to $x_2(t+\Delta t)$
2. Estimate the truncation error by
$$\Delta x(\Delta t) = x_2(t+\Delta t) - x_1(t+\Delta t) \approx \mathcal{O}\left((\Delta t)^5\right)$$
3. Return $x_2(t+\Delta t)$ as an answer, because that's going to be the more accurate one.
4. Since $\Delta x \approx \mathcal{O}\left((\Delta t)^5\right)$ assuming that we try two different values of Δt we should have:
$$\frac{(\Delta x)_0}{(\Delta x)_1} = \left(\frac{(\Delta t)_0}{(\Delta t)_1}\right)^5$$
   This yields the following formula for a step size:
$$(\Delta t)_0 = (\Delta t)_1 \left|\frac{(\Delta x)_0}{(\Delta x)_1}\right|^{1/5} \tag{4.84}$$

Strategy Let $(\Delta x)_0$ be the *requested* accuracy.

- If $(\Delta x)_1 > (\Delta x)_0$ equation (4.84) tells us how much to reduce the stepsize when we *repeat* the *failed* step.
- If $(\Delta x)_1 < (\Delta x)_0$ equation (4.84) tells us how much we can stretch the stepsize for the *next* step.

4.4.3 Example Code

```
SUBROUTINE rkqs(y, dydx, x, htry, eps, yscal, hdid, hnext, derivs)

  USE nrtype; USE nrutil, ONLY : assert_eq, nrerror
  USE nr, ONLY : rkck

  IMPLICIT NONE

  REAL(sp), DIMENSION(:), INTENT(inout) :: y
  REAL(sp), DIMENSION(:), INTENT(in) :: dydx, yscal
  REAL(sp), INTENT(inout) :: x
  REAL(sp), INTENT(in) :: htry, eps
  REAL(sp), INTENT(out) :: hdid, hnext

  INTERFACE
     SUBROUTINE derivs(x, y, dydx)
```

```
      USE nrtype
      IMPLICIT NONE
      REAL(sp), INTENT(in) :: x
      REAL(sp), DIMENSION(:), INTENT(in) :: y
      REAL(sp), DIMENSION(:), INTENT(out) :: dydx
    END SUBROUTINE derivs
  END INTERFACE

  INTEGER(i4b) :: ndum
  REAL(sp) :: errmax, h, htemp, xnew
  REAL(sp), DIMENSION(SIZE(y)) :: yerr, ytemp
  REAL(sp), PARAMETER :: safety=0.9_sp, pgrow=-0.2_sp, pshrnk=-0.25_sp, &
      errcon=1.89e-4 ! (5/safety)**(1/pgrow)

  ndum=assert_eq(SIZE(y), SIZE(dydx), SIZE(yscal), 'rkqs')
  h=htry
  DO
     CALL rkck(y, dydx, x, h, ytemp, yerr, derivs)
     errmax=MAXVAL(ABS(yerr(:)/yscal(:)))/eps
     IF (errmax <= 1.0) EXIT
     htemp=safety*h*(errmax**pshrnk)
     h = SIGN(MAX(ABS(htemp), 0.1_sp*ABS(h)), h)
     xnew = x+h
     IF (xnew == x) CALL nrerror('stepsize underflow in rkqs')
  END DO
  IF (errmax > errcon) THEN
     hnext=safety*h*(errmax**pgrow)
  ELSE
     hnext=5.0_sp*h
  END IF
  hdid=h
  x=x+h
  y(:)=ytemp(:)

END SUBROUTINE rkqs
```

Discussion

This code is really just a wrapper.

The variables are similar to the ones used in the previous Runge Kutta subroutine:

y the equation that is being solved is of the form

$$\frac{\mathrm{d}\boldsymbol{y}(x)}{\mathrm{d}x} = \mathrm{derivs}(x, \boldsymbol{y})$$

so this variable represents a vector of y_k values, where k runs through the equations. On entry $\boldsymbol{y}$ stands for $\boldsymbol{y}(x_n)$. On exit it will stand for $\boldsymbol{y}(x_n + \Delta x)$.

dydx is the initial value of derivs$(x, \boldsymbol{y})$, that is derivs$(x_n, \boldsymbol{y}(x_n))$.

x is x_n

htry is the initial guess for a good Δx that may be changed if a requested accuracy of integration is not met.

eps The error associated with the whole system of equations is going to be evaluated in the following manner. The Runge-Kutta subroutine `rkck` to be called by `rkqs` will return a new value $\boldsymbol{y}(x_n + \Delta x)$ plus an absolute error estimate $\Delta\boldsymbol{y}$. That error is then going to be *scaled* by dividing by a user supplied array `yscal`. This new *scaled* error is then going to be compared to ϵ, which is what this parameter `eps` stands for.

yscal is the scaling array. If you're happy with unscaled values of $\boldsymbol{y}$ simply set it to 1.

hdid is the value of Δx that has really been used, after all the mucking up, to make the step.

hnext is the suggested next value of Δx.

derivs is the subroutine used to evaluate the right hand side of

$$\frac{\mathrm{d}\boldsymbol{y}(x)}{\mathrm{d}x} = \boldsymbol{f}(x, \boldsymbol{y})$$

As most other codes discussed in this lecture notes the action here begins by checking that the dimensions of variables passed to the subroutine are correct. The dimension of $\boldsymbol{y}$ that is extracted is discarded:

```
ndum=assert_eq(SIZE(y), SIZE(dydx), SIZE(yscal), 'rkqs')
```

Then the step size Δx is set to the suggested step size of `htry`:

```
h=htry
```

and we enter the `DO` loop within which we

1. call `rkck` to evaluate the next $\boldsymbol{y}$ for the suggested value of Δx, but also to give us the truncation error in `yerr`:

   ```
   CALL rkck(y, dydx, x, h, ytemp, yerr, derivs)
   ```

2. scale the returned error and compare it to ϵ:

   ```
   errmax=MAXVAL(ABS(yerr(:)/yscal(:)))/eps
   IF (errmax <= 1.0) EXIT
   ```

   observe that since we are working with the system of equations here, and each equation is going to have its own different value of error Δy_k, we pick up the largest error and make that stand for the error for the whole system of equations. If the error is within the prescribed limits we accept the result and exit the `DO` loop.

3. If the error is too large, then we have to repeat the step with a shorter Δx. So we shrink the step, but by no more than a factor of 10:

   ```
   htemp=safety*h*(errmax**pshrnk)
   h = SIGN(MAX(ABS(htemp), 0.1_sp*ABS(h)), h)
   ```

4. then check if the step size hasn't shrunk too much, in which case the subroutine aborts with an error message:

   ```
   xnew = x+h
   IF (xnew == x) CALL nrerror('stepsize underflow in rkqs')
   ```

5. and return to the top of the loop when the next trial Runge Kutta step is made.

After we have finished with the looping and have the new values of $\boldsymbol{y}$ as well as the new value of error, we suggest stretching Δx, but by no more than 5 times:

```
IF (errmax > errcon) THEN
   hnext=safety*h*(errmax**pgrow)
ELSE
   hnext=5.0_sp*h
END IF
```

The last three lines move the used Δx into `hdid`, the new value of $\boldsymbol{y}$ goes into `y`, and x itself becomes updated to $x + \Delta x$:

```
hdid=h
x=x+h
y(:)=ytemp(:)
```

The subroutine `rkck` is going to be very similar to subroutine `rk4`, but instead of making just one Runge-Kutta step, it has to make

1. one full size step
2. two half size steps

then compare the results and return the result of the two half size steps in `ytemp` and the observed error in `yerr`.

I leave it to you to develop your own version of `rkck`. The easiest way to do that is to rewrite subroutine `rk4`, which we have discussed in section 4.4.1.

4.5 The Bulirsch-Stoer Method

There are two major components that form a foundation of the Bulirsch-Stoer Method. Both are very useful in their own right, and thus warrant an extended discussion. They are:

1. the modified midpoint method
2. Richardson extrapolation

4.5.1 The Modified Midpoint Method

The Modified Midpoint Method is based on the following formulas:

$$\begin{aligned}
x(t_0) &= x_0 \\
x_1 = x(t_1) &= x_0 + f(x_0, t_0)\Delta t \quad t_1 = t_0 + \Delta t \\
x_2 = x(t_2) &= x_0 + f(x_1, t_1)2\Delta t \quad t_2 = t_1 + \Delta t \\
x_3 = x(t_3) &= x_1 + f(x_2, t_2)2\Delta t \quad t_3 = t_2 + \Delta t \\
\ldots \\
x_{m+1} = x(t_{m+1}) &= x_{m-1} + f(x_m, t_m)2\Delta t \quad t_m = t_{m-1} + \Delta t
\end{aligned}$$

This method is also called a leap-frog method and is often used in particle codes, being quite a lot cheaper than the Runge-Kutta method, while at the same time, offering a better stability than the Euler method.

The following code illustrates Fortran-90 implementation:

```
SUBROUTINE mmid(y, dydx, xs, htot, nstep, yout, derivs)

  USE nrtype; USE nrutil, ONLY: assert_eq, swap
  IMPLICIT NONE
  INTEGER(i4b), INTENT(in) :: nstep
  REAL(sp), INTENT(in) :: xs, htot
  REAL(sp), DIMENSION(:), INTENT(in) :: y, dydx
  REAL(sp), DIMENSION(:), INTENT(out) :: yout
  INTERFACE
     SUBROUTINE derivs(x, y, dydx)
       USE nrtype
       IMPLICIT NONE
       REAL(sp), INTENT(in) :: x
       REAL(sp), DIMENSION(:), INTENT(in) :: y
       REAL(sp), DIMENSION(:), INTENT(out) :: dydx
     END SUBROUTINE derivs
  END INTERFACE
  INTEGER(i4b) :: n, ndum
  REAL(sp) :: h, h2, x
  REAL(sp), DIMENSION(SIZE(y)) :: ym, yn

  ndum = assert_eq(SIZE(y), SIZE(dydx), SIZE(yout), 'mmid')
  h = htot / nstep
  ym = y
  yn = y + h * dydx
  x = xs + h
  CALL derivs(x, yn, yout)
  h2 = 2.0_sp * h
  DO n = 2, nstep
     CALL swap(ym, yn)
     yn = yn + h2 * yout
     x = x + h
     CALL derivs(x, yn, yout)
  END DO
  yout = 0.5_sp * (ym + yn + h * yout)

END SUBROUTINE mmid
```

The Discussion

This is a very simple, almost a self-explanatory code. I'll skip most of the variable definitions and the `assert_eq` bit, because it's much the same as before.

xs is the starting point, x_0

htot is the total step Δx to be taken, but it's going to be split into multiple shorter steps

nstep is the number of those shorter steps to be used

We begin by evaluating the length of a mid-point step by dividing our input value of Δx by `nstep`:

```
h = htot / nstep
```

Thus $h = \Delta x/n$. Then we set $\boldsymbol{y}_m$, to $\boldsymbol{y}$ and $\boldsymbol{y}_n$ to $\boldsymbol{y} + \mathrm{dydx} h$ and set $x = x_0 + h$:

```
ym = y
yn = y + h * dydx
x = xs + h
```

So that at this stage $\boldsymbol{y}_m = \boldsymbol{y}_0$ and $\boldsymbol{y}_n = \boldsymbol{y}_1$, whereas $x = x_1$.

The next step is to evaluate the right hand side of the differential equation at x_1 and $\boldsymbol{y}_1$:

```
CALL derivs(x, yn, yout)
h2 = 2.0_sp * h
```

Now we are ready to start the leapfrog. h_2 becomes $2h$ and we enter the `DO` loop within which:

1. we swap the roles of $\boldsymbol{y}_m$ and $\boldsymbol{y}_n$, so that the guy that has become an endpoint as the result of the last step, becomes the midpoint point now, whereas the guy that was a midpoint before becomes a starting point now. Time, i.e., x, in the meantime becomes advanced by half-step, i.e., h:

   ```
   CALL swap(ym, yn)
   yn = yn + h2 * yout
   x = x + h
   ```

2. having gone through these preparations we evaluate the right hand side of the equation at the new end-point that is going to become a mid-point for the next jump:

   ```
   CALL derivs(x, yn, yout)
   ```

The last step, just before leaving the subroutine, is to wind down the process by taking the average of the *implicitly* evaluated last value of $\boldsymbol{y}$, i.e., `ym + h * yout` and *explicitly* evaluated value, i.e., `yn`, and return that average in `yout`:

```
yout = 0.5_sp * (ym + yn + h * yout)
```

4.5.2 Richardson Interpolation and Extrapolation

```
SUBROUTINE polint(xa, ya, x, y, dy)

  USE nrtype; USE nrutil, ONLY : assert_eq, iminloc, nrerror
  IMPLICIT NONE
  REAL(sp), DIMENSION(:), INTENT(in) :: xa, ya
  REAL(sp), INTENT(in) :: x
  REAL(sp), INTENT(out) :: y, dy

  ! Given arrays xa and ya of length N, and given a value x, this routine
  ! returns a value y, and an error estimate dy. If P(x) is the polynomial
  ! of degree N - 1 such that P(x a_i) = y a_i, i = 1, ..., N, then the
  ! returned value y = P(x).

  INTEGER(i4b) :: m, n, ns
  REAL(sp), DIMENSION(SIZE(xa)) :: c, d, den, ho

  n = assert_eq(SIZE(xa), SIZE(ya), 'polint')
  c = ya                         ! Initialize the tableau of c's and d's
  d = ya
  ho = xa - x
  ns = iminloc(ABS(x - xa))      ! Find index ns of closest table entry
  y = ya(ns)                     ! Initial approximation to y.
  ns = ns - 1
  DO m = 1, n - 1                       ! For each column of the tableau
     den(1:n-m) = ho(1:n-m) - ho(1+m:n)  ! we loop over c's and d's and
     IF (ANY(den(1:n-m) == 0.0)) &       ! update them
          CALL nrerror('polint: calculation failure')

     ! This error can occur only if two input xa's are (to within roundoff)
     ! identical.

     den(1:n - m) = (c(2:n-m+1) - d(1:n-m))/den(1:n-m)
     d(1:n-m) = ho(1+m:n) * den(1:n-m)   ! Here c's and d's get updated
     c(1:n-m) = ho(1:n-m) * den(1:n-m)
     IF (2 * ns < n-m) THEN      ! After each column in the tableau is
        dy=c(ns+1)               ! completed decide, which correction
     ELSE                        ! c or d we add to y. We take the
        dy=d(ns)                 ! straightest line through the tableau
        ns=ns-1                  ! to its apex. The partial approximations
     END IF                      ! are thus centred on x. The last dy
     y = y+dy                    ! is the measure of error.
  END DO

END SUBROUTINE polint


SUBROUTINE ratint(xa, ya, x, y, dy)

  USE nrtype; USE nrutil, ONLY : assert_eq, iminloc, nrerror
  IMPLICIT NONE
  REAL(sp), DIMENSION(:), INTENT(in) :: xa, ya
  REAL(sp), INTENT(in) :: x
  REAL(sp), INTENT(out) :: y, dy

  ! Given arrays xa and ya of length N, and given a value of x, this routine
  ! returns a value of y and an accuracy estimate dy. The value returned is
  ! that of the diagonal rational function, evaluated at x, that passes
  ! through the N points (xa_i, ya_i), i = 1... N.

  INTEGER(i4b) :: m, n, ns
  REAL(sp), DIMENSION(SIZE(xa)) :: c, d, dd, h, t
  REAL(sp), PARAMETER :: tiny=1.0e-25_sp

  n = assert_eq(SIZE(xa), SIZE(ya), 'ratint')
  h = xa - x
  ns = iminloc(ABS(h))
  y = ya(ns)
  IF (x == xa(ns)) THEN
     dy = 0.0
     RETURN
  END IF
  c = ya
  d = ya + tiny               ! The tiny is needed to prevent 0/0
  ns = ns - 1
  DO m=1, n-1
     t(1:n-m) = (xa(1:n-m)-x) * d(1:n-m)/h(1+m:n)   ! h will never be 0
     dd(1:n-m) = t(1:n-m) - c(2:n-m+1)
     IF (ANY(dd(1:n-m) == 0.0)) &                   ! interpolating function
          CALL nrerror('failure in ratint')         ! has a pole here
     dd(1:n-m) = (c(2:n-m+1) - d(1:n-m))/dd(1:n-m)
     d(1:n-m) = c(2:n-m+1) * dd(1:n-m)
     c(1:n-m) = t(1:n-m) * dd(1:n-m)
     IF(2*ns < n-m) THEN
        dy = c(ns+1)
     ELSE
        dy = d(ns)
        ns = ns-1
     END IF
     y=y+dy
  END DO

END SUBROUTINE ratint


SUBROUTINE pzextr(iest, xest, yest, yz, dy)

  USE nrtype; USE nrutil, ONLY : assert_eq, nrerror
  IMPLICIT NONE
  INTEGER(i4b), ikntent(in) :: iest
  REAL(sp), INTENT(in) :: xest
  REAL(sp), DIMENSION(:), INTENT(in) :: yest
  REAL(sp), DIMENSION(:), INTENT(out) :: yz, dy

  ! Use polynomial extrapolation to evaluate N functions at x = 0 by fitting
  ! a polynomial to a sequence of estimates with progressively smaller
  ! values x = xest, and corresponding function vectors yest. This call
  ! is number iest in the sequence of calls. Extrapolated function values
  ! are output as yz, and their estimated error is output as dy. yest, yz,
  ! and dy are arrays of length N.

  INTEGER(i4b), PARAMETER :: iest_max = 16
  INTEGER(i4b) :: j, nv
  INTEGER(i4b), SAVE :: nvold = -1
  REAL(sp) :: delta, f1, f2
  REAL(sp), DIMENSION(SIZE(yz)) :: d, tmp, q
  REAL(sp), DIMENSION(iest_max), SAVE :: x
  REAL(sp), DIMENSION(:,:), ALLOCATABLE, SAVE :: qcol

  nv = assert_eq(SIZE(yz), SIZE(yest), SIZE(dy), 'pzextr')
  IF (iest > iest_max) &
       CALL nrerror('pzextr: probable misuse, too much extrapolation')
  IF(nv /= nvold) THEN      ! set up internal storage
     IF(ALLOCATED(qcol)) DEALLOCATE(qcol)
     ALLOCATE(qcol(nv, iest_max))
     nvold=nv
  END IF
  x(iest) = xest            ! save current independent variable
  dy(:) = yest(:)
  yz(:) = yest(:)
  IF (iest == 1) THEN       ! store first estimate in first column
     qcol(:,1)=yest(:)
  ELSE
     d(:) = yest(:)
     DO j=1, iest-1
        delta=1.0_sp/(x(iest-j)-xest)
        f1=xest*delta
        f2=x(iest-j)*delta
        q(:) = qcol(:,j)    ! propagate tableau 1 diagonal more
        qcol(:,j)=dy(:)
        tmp(:)=d(:)-q(:)
        dy(:)=f1*tmp(:)
        d(:)=f2*tmp(:)
        yz(:)=yz(:)+dy(:)
     END DO
     qcol(:,iest)=dy(:)
  END IF

END SUBROUTINE pzextr


SUBROUTINE rzextr(iest, xest, yest, yz, dy)

  USE nrtype; USE nrutil, ONLY : assert_eq, nrerror
  IMPLICIT NONE
  INTEGER(i4b), INTENT(in) :: iest
  REAL(sp), INTENT(in) :: xest
  REAL(sp), DIMENSION(:), INTENT(in) :: yest
  REAL(sp), DIMENSION(:), INTENT(out) :: yz, dy

  ! Exact substitute for pzextr, but uses diagonal rational function
  ! extrapolation instead of polynomial extrapolation.

  INTEGER(i4b), PARAMETER :: iest_max = 16
  INTEGER(i4b) :: k, nv
  INTEGER(i4b), SAVE :: nvold = -1
  REAL(sp), DIMENSION(SIZE(yz)) :: yy, v, c, b, b1, ddy
  REAL(sp), DIMENSION(:,:), ALLOCATABLE, SAVE :: d
  REAL(sp), DIMENSION(iest_max), SAVE :: fx, x

  nv = assert_eq(SIZE(yz), SIZE(dy), SIZE(yest), 'rzextr')
  IF(iest > iest_max) &
       CALL nrerror('rzextr: probable misuse, too much extrapolation')
  IF (nv /= nvold) THEN
     IF(ALLOCATED(d)) DEALLOCATE(d)
     ALLOCATE(d(nv, iest_max))
     nvold=nv
  END IF
  x(iest)=xest                 ! save current independent variable
  IF(iest == 1) THEN
     yz=yest
     d(:,1)=yest
     dy=yest
  ELSE
     fx(2:iest)=x(iest-1:1:-1)/xest
     yy=yest                   ! evaluate next diagonal in tableau
     v=d(1:nv, 1)
     c=yy
     d(1:nv, 1)=yy
     DO k=2, iest
        b1=fx(k)*v
        b=b1-c
        WHERE(b/=0.0)
           b=(c-v)/b
           ddy=c*b
           c=b1*b
        ELSEWHERE              ! care needed to avoid division by 0
           ddy=v
        END WHERE
        IF (k/=iest) v=d(1:nv, k)
        d(1:nv, k) = ddy
        yy=yy+ddy
     END DO
     dy=ddy
     yz=yy
  END IF

END SUBROUTINE rzextr
```

4.5.3 Bulirsch-Stoer Step

```
SUBROUTINE bsstep(y, dydx, x, htry, eps, yscal, hdid, hnext, derivs)

  USE nrtype; USE nrutil, ONLY : arth, assert_eq, cumsum, iminloc, nrerror, &
       outerdiff, outerprod, upper_triangle
  USE nr, ONLY : mmid, pzextr
  IMPLICIT NONE
  REAL(sp), DIMENSION(:), INTENT(inout) :: y
  REAL(sp), DIMENSION(:), INTENT(in) :: dydx, yscal
  REAL(sp), INTENT(inout) :: x
  REAL(sp), INTENT(in) :: htry, eps
  REAL(sp), INTENT(out) :: hdid, hnext
  INTERFACE
     SUBROUTINE derivs(x, y, dydx)
       USE nrtype
       IMPLICIT NONE
       real9sp), INTENT(in) :: x
       REAL(sp), DIMENSION(:), INTENT(in) :: y
       REAL(sp), DIMENSION(:), INTENT(out) :: dydx
```

```
   END SUBROUTINE derivs
END INTERFACE
INTEGER(i4b), PARAMETER :: imax = 9, kmaxx=imax-1
REAL(sp), PARAMETER :: safe1=0.25_sp, safe2=0.7_sp, redmax=1.0e-5_sp, &
     redmin=0.7_sp, tiny=1.0e-30_sp, scalmx=0.1_sp

! Bulirsch-Stoer step with monitoring of local truncation error to
! ensure accuracy and adjust stepsize. Input are athe dependent variable
! vector y and its derivative dydx at the starting value of the independent
! variable x. Also input are the stepsize that was actually accomplished,
! and hnext is the estimated next stepsize. derivs is the user-supplied
! subroutine that computes the right-hand-side derivatives. y, dydx, and
! yscal must all have the same length. Be sure to set htry on successive
! steps to the value of hnext returned from the previous step, as is the
! case if the routine is called by odeint.
! Parameters: kmaxx is the maximum row number used in the extrapolation;
! imax is the next row number; safe1 and safe2 are safety factors;
! redmax is the maximum factor used when a stepsize is reduced, redmin
! the minimum; tiny prevents division by zero; 1/scalmx is the maximum
! factor by which a stepsize can be increased.

INTEGER(i4b) :: k, km, ndum
INTEGER(i4b, DIMENSION(imax) :: nseq = (/ 2, 4, 6, 8, 10, 12, 14, 16, 18 /)
INTEGER(i4b), SAVE :: kopt, kmax
REAL(sp), DIMENSION(kmaxx, kmaxx), SAVE :: alf
REAL(sp), DIMENSION(kmaxx) :: err
REAL(sp), DIMENSION(imax), SAVE :: a
REAL(sp), SAVE :: epsold = -1.0_sp, xnew
REAL(sp) :: eps1, errmax, fact, h, red, scale, wrkmin, xest
REAL(sp), DIMENSION(SIZE(y)) :: yerr, ysav, yseq
LOGICAL(lgt) :: reduct
LOGICAL(lgt, SAVE :: first = .TRUE.

ndum = assert_eq(SIZE(y), SIZE(dydx), SIZE(yscal), 'bsstep')
IF (eps /= epsold) THEN             ! a new tolerance, reinitialize
   hnext = -1.0e29_sp               ! "impossible" values
   xnew=-1.0e29_sp
   eps1=safe1*eps
   a(:)=cumsum(nseq,1)
   WHERE (upper_triangle(kmaxx, kmaxx)) alf=eps1** &
        (outerdiff(a(2:), a(2:))/outerprod(arth( &
        3.0_sp, 2.0_sp, kmaxx), (a(2:)-a(1)+1.0_sp)))
   epsold=eps
   DO kopt=2,kmaxx-1                ! determine optimal row number for
      IF (a(kopt+1) > a(kopt)*alf(kopt-1,kopt)) EXIT      ! convergence
   END DO
   kmax=kopt
END IF
h = htry
ysav(:) = y(:)                      ! save the starting values
IF (h /= hnext .OR. x /= xnew) THEN  ! a new stepsize or a new integration
   first = .TRUE.                   ! re-establish the order window
   kopt = kmax
END IF
reduct = .FALSE.
main_loop: DO
   DO k = 1, kmax                   ! evaluate the sequence of modified
```

```
      xnew = x+h                     ! midpoint integrations
      IF (xnew == x) CALL nrerror('step size underflow in bsstep')
      CALL mmid(ysav, dydx, x, h, nseq(k), yseq, derivs)
      xest=(h/nseq(k))**2            !squared, since error series is even
      CALL pzextr(k, xest, yseq, y, yerr) ! perform extrapolation
      IF (k /= 1) THEN                ! computer normalized error estimate
         errmax=MAXVAL(ABS(yerr(:)/yscal(:)))
         errmax=MAX(tiny, errmax)/eps   ! scale error relative to tolerance
         km=k-1
         err(km)=(errmax/safe1)**(1.0_sp/(2*km+1))
      END IF
      IF (k /= 1 .AND. (k >= kopt - 1 .OR. first)) THEN  ! in order window
         IF (errmax < 1.0) EXIT main_loop      ! converged
         IF(k == kmax .OR. k == kopt+1) THEN   ! check for possible step
            red = safe2/err(km)                ! size reduction
            EXIT
         ELSE IF (k == kopt) THEN
            IF (alf(kopt-1, kopt) < err(km)) THEN
               red=1.0_sp/err(km)
               EXIT
            END IF
         ELSE IF (kopt == kmax) THEN
            IF(alf(km, kmax-1) < err(km)) THEN
               red = alf(km, kmax-1) * safe2/err(km)
               EXIT
            END IF
         ELSE IF (alf(km, kopt) < err(km)) THEN
            red=alf(km,kopt=1) / err(km)
            EXIT
         END IF
      END IF
   END DO
   red=MAX(MIN(red, redmin), redmax)   ! reduce step size by at least
   h = h*red                           ! redmin and at most redmax
   reduct = .TRUE.
 END DO main_loop                      ! try again
 x = xnew                              ! successful step taken
 hdid=h
 first=.FALSE.
 kopt=1+iminloc(a(2:km+1)*MAX(err(1:km), scalmx))

! Compute optimal row for convergence and corresponding stepsize

 scale=MAX(err(kopt=1), scalmx)
 wrkmin=scale*a(kopt)
 hnext=h/scale
 IF(kopt >= k .AND. kopt /= kmax .AND. .NOT. reduct) THEN ! check for possible
    fact = MAX(scale/alf(kopt-1, kopt), scalmx)           ! order increase
    IF(a(kopt+1)*fact <=wrkmin) THEN                      ! but not if step
       hnext = h/fact                                     ! size just reduced
       kopt=kopt+1
    END IF
 END IF

END SUBROUTINE bsstep
```

4.6 Stiff Equations

- thin boundary layers
- p-n junctions

Every time we encounter a problem where things change on two vastly different scales, the equations get *stiff*.

But even innocent ODEs of second order can easily lead to stiff systems of first order ODEs.

Consider the following example:

$$\frac{\mathrm{d}^2 x(t)}{\mathrm{d}t^2} = 100x \tag{4.85}$$

At first glance there is just one scale in this equation, which is 100. Now assume a solution of the following form:

$$x(t) = Ae^{at} \tag{4.86}$$

then

$$\begin{aligned} \frac{\mathrm{d}x(t)}{\mathrm{d}t} &= aAe^{at} \\ \frac{\mathrm{d}^2 x(t)}{\mathrm{d}t^2} &= a^2 Ae^{at} = a^2 x(t) \end{aligned}$$

substituting this solution into our equation yields

$$a = \pm 10 \tag{4.87}$$

The most general solution is therefore a linear combination of two possibilities:

$$x(t) = Ae^{10t} + Be^{-10t} \tag{4.88}$$

There are two vastly different time scales in this solution.

Even if you set your initial conditions so that $A = 0$, e.g.,

$$\begin{aligned} x(0) &= 1 \\ \left.\frac{\mathrm{d}x(t)}{\mathrm{d}t}\right|_{t=0} &= -10 \end{aligned}$$

rounding errors will add a small amount of the e^{10t} solution, so that numerically you'll get:

$$x(t) = e^{-10t} + \epsilon e^{10t} \tag{4.89}$$

and if you integrate long enough, the solution will eventually blow up.

We had seen it already when we talked about the instability of the *explicit* Euler method.

Here is how you can *stabilize* the solution method *regardless* of the length of your time step, sic!

Consider the following very simple equation:

$$\frac{\mathrm{d}x(t)}{\mathrm{d}t} = -cx(t) \tag{4.90}$$

The derivative itself can be approximated by:

$$\frac{\mathrm{d}x(t)}{\mathrm{d}t} \approx \frac{x(t+\Delta t) - x(t)}{\Delta t} \tag{4.91}$$

In the Euler explicit scheme the right hand side was evaluated at $x(t)$. But we can just as well evaluate it at $x(t+\Delta t)$. This leads to the following equation:

$$\frac{x(t+\Delta t) - x(t)}{\Delta t} = -cx(t+\Delta t) \tag{4.92}$$

and the solution is:

$$x(t+\Delta t) = \frac{x(t)}{1+c\Delta t} \tag{4.93}$$

This solution is *always* going to converge to zero, regardless of the step length Δt. If Δt is too large, the convergence may not be very accurate, but the numerical solution itself is not going to explode.

This solution method is said to be *unconditionally* stable.

The reasoning can be easily expanded to a system of ODEs as follows. Consider the following:

$$\frac{\mathrm{d}\boldsymbol{x}(t)}{\mathrm{d}t} = -\boldsymbol{C}\cdot\boldsymbol{x}(t) \tag{4.94}$$

Implicit differencing yields

$$\boldsymbol{x}(t+\Delta t) = (\mathbf{1}+\boldsymbol{C}\Delta t)^{-1}\cdot\boldsymbol{x}(t) \tag{4.95}$$

This equation can be solved analytically, for example, by diagonalizing $\boldsymbol{C}$. In the corresponding base the solution can be then written as:

$$\tilde{\boldsymbol{x}}_i(t+\Delta t) = \frac{\tilde{\boldsymbol{x}}_i(t)}{1+\lambda_i\Delta t} \tag{4.96}$$

In order to obtain $\boldsymbol{x}(t+\Delta t)$ we have to rotate it back to the original basis.

We have discussed this step in P573.

If all λ_i are positive or zero (i.e., $\boldsymbol{C}$ is said to be *positive definite*) then the method is stable for any step size Δt.

In order to solve the equation we can invert $1+\boldsymbol{C}\Delta t$ at the first step, and then, assuming that Δt is fixed, simply keep reusing it, because $\boldsymbol{C}$ is constant.

If $\boldsymbol{C} = \boldsymbol{C}(t)$ but still definite positive for every t, or if we're going to change Δt, we may have to invert $1+\boldsymbol{C}\Delta t$ at every step, or to invoke one of our eigen-value procedures to do the job – and then rotate the solution as need be.

In general

$$\frac{\mathrm{d}\boldsymbol{x}(t)}{\mathrm{d}t} = \boldsymbol{f}(\boldsymbol{x}, t) \tag{4.97}$$

and the implicit differentiation scheme:

$$\boldsymbol{x}(t + \Delta t) = \boldsymbol{x}(t) + \Delta t \boldsymbol{f}(\boldsymbol{x}(t + \Delta t), t + \Delta t) \tag{4.98}$$

leads to a set of nonlinear algebraic equations that have to be solved iteratively, e.g., using the Newton-Raphson method, at every time step. In general case this scheme may *not* guaranteed to be stable, but, if the Jacobian $\partial \boldsymbol{f}/\partial \boldsymbol{x}$ is positive definite for every t then the method is stable.

If the time step Δt is a short one, then

$$\boldsymbol{f}(\boldsymbol{x}(t + \Delta t), t + \Delta t) \approx \boldsymbol{f}(\boldsymbol{x}(t), t + \Delta t) + \left.\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}\right|_{\boldsymbol{x}=\boldsymbol{x}(t)} \cdot (\boldsymbol{x}(t + \Delta t) - \boldsymbol{x}(t)) \tag{4.99}$$

and the solution to this equation is

$$\boldsymbol{x}(t{+}\Delta t) = \left(\mathbf{1} - \Delta t \left.\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}\right|_{\boldsymbol{x}=\boldsymbol{x}(t)}\right)^{-1} \cdot \left(\boldsymbol{x}(t) + \Delta t \left(\boldsymbol{f}(\boldsymbol{x}(t), t + \Delta t) - \left.\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}\right|_{\boldsymbol{x}=\boldsymbol{x}(t)} \cdot \boldsymbol{x}(t)\right)\right) \tag{4.100}$$

So the only matrix we have to invert in this case is:

$$\mathbf{1} - \Delta t \left.\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}\right|_{\boldsymbol{x}=\boldsymbol{x}(t)} \tag{4.101}$$

This is called a *semi-implicit* method and its stability depends on the sign of the Jacobian $\partial \boldsymbol{f}/\partial \boldsymbol{x}$, as in the full implicit case.

The implicit and semi-implicit methods discussed so far are, like the Euler method, first-order accurate only. Higher order implicit methods exist too, but they are prohibitively costly and thus seldom used. It is usually cheaper to shorten the time step while enjoying the stability of a first-order implicit or semi-implicit method at the same time. For a longer time step a semi-implicit method may break to begin with, so the resulting cost would be additionally compounded by having to go back to a fully non-linear case.

Semi-implicit methods are used commonly whenever radiative transfer has to be combined with CFD computations. The reason for this is a very short time scale associated with radiative transfer phenomena compared to CFD time scales. Weather prediction codes and ocean-atmosphere circulation codes are the place where you'll encounter such techniques. Another area is simulation of astrophysical systems, such as accretion disks, supernova explosions, and collisions between neutron stars.

Implicit and semi-implicit methods applied to systems of ODEs and PDEs (the latter can be reduced to the former) require the use of matrix inversion methods, or, at the very least, the use of linear equation solvers. This is what we are going to focus on in the next chapter.

Chapter 5

Other Matrix Operations

5.1 Gauss-Jordan Elimination

Consider the following equation:

$$\boldsymbol{A} \cdot \boldsymbol{x} = \boldsymbol{b} \tag{5.1}$$

The equation can be solved as follows:

1. divide the first row by A_{11}, so that the new A_{11} becomes 1
2. subtract from the second row A_{2i} the first row multiplied by A_{21}, so that after that subtraction A_{21} becomes 0
3. subtract from the third row A_{3i} the first row multiplied by A_{31}, so that after that subtraction A_{31} becomes 0
4. ...
5. subtract from the n^{th} row A_{ni} the first row multiplied by A_{n1}, so that after that subtraction A_{n1} becomes 0. Now the first column of matrix A is
$$\begin{pmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$
6. divide the second row by A_{22}, so that the new A_{22} becomes 1
7. subtract from the first row A_{1i} the second row multiplied by A_{12}, so that after that subtraction A_{12} becomes 0
8. subtract from the third row A_{3i} the second row multiplied by A_{32}, so that after that subtraction A_{32} becomes 0

9. subtract from the fourth row A_{4i} the second row multiplied by A_{42}, so that after that subtraction A_{42} becomes 0
10. ...
11. subtract from the n^{th} row A_{ni} the second row multiplied by A_{n2}, so that after that subtraction A_{n2} becomes 0. Now the first *two* columns of matrix A
$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ \vdots & \vdots \\ 0 & 0 \end{pmatrix}$$
12. ...

At the same time matching operations must be performed on vector $\boldsymbol{b}$.

When the process is finished we get the following new equation

$$\begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{pmatrix} \tag{5.2}$$

The coefficients b_i are quite different now, but the equation can be solved trivially. And so, we get:

$$x_n = b_n \tag{5.3}$$

In matrix notation the operations performed on $\boldsymbol{A}$ amount to having found such matrix $\boldsymbol{A}^{-1}$ that

$$\boldsymbol{A}^{-1} \cdot \boldsymbol{A} \cdot \boldsymbol{x} = \boldsymbol{1} \cdot \boldsymbol{x} = \boldsymbol{A}^{-1} \cdot \boldsymbol{b} \tag{5.4}$$

Now, if during the computation we were to perform all the motions not only on vector $\boldsymbol{b}$, but also on another matrix, which has been initialized to $\boldsymbol{1}$, we would end up with $\boldsymbol{A}^{-1}$ inside that matrix when the whole thing is over.

When these computations are carried out solutions can be found *simultaneously* to systems of equations with various right hand sides (but always the same left hand side $\boldsymbol{A}$ so vectors $\boldsymbol{b}$ are often aligned into a matrix, which doesn't have to be square, and that matrix is then also accompanied by a square matrix that has been initialized to $\boldsymbol{1}$, so as to yield $\boldsymbol{A}^{-1}$ as well.

Although the above comprises the heart of the method there is one complication that we have to incorporate. It may happen that a particular diagonal term A_{kk} is zero or very small. In that case dividing whatever's left of A_{ki} by A_{kk} may lead to overflows. If this is the case then the solution is to interchange the rows or the columns so as to place the *largest* element of A_{ki}, which is called a *pivot* in the A_{kk} position, and get the small one in the pivot's old location. This is an essential part of the Gauss-Jordan Elimination technique and the program must *never* be written without pivoting.

5.1.1 Example Program

Here is an example program:

```
SUBROUTINE gaussj(a, b)

  USE nrtype; USE nrutil, ONLY : assert_eq, nrerror, outerand, outerprod, swap
  IMPLICIT NONE
  REAL(sp), DIMENSION(:,:), INTENT(inout) :: a, b

  ! Linear equation solution by Gauss-Jordan elimination.
  ! a is an NxN input coefficient matrix. b is an NxM input matrix
  ! containing M right-hand-side vectors. On output, a is replaced
  ! by its matrix inverse, and b is replaced by the corresponding
  ! set of solution vectors.

  INTEGER(i4b), DIMENSION(SIZE(a, 1)) :: ipiv, indxr, indxc

  ! These arrays are used for bookkeeping on the pivoting

  LOGICAL(lgt), DIMENSION(SIZE(a, 1)) :: lpiv
  REAL(sp) :: pivinv
  REAL(sp), DIMENSION(SIZE(a, 1)) :: dumc
  INTEGER(i4b), TARGET :: irc(2)
  INTEGER(i4b) :: i, l, n
  INTEGER(i4b), POINTER :: irow, icol

  n = assert_eq(SIZE(a, 1), SIZE(a, 2), SIZE(b, 1), 'gaussj')
  irow => irc(1)
  icol => irc(2)
  ipiv = 0
  DO i = 1, n
     lpiv = (ipiv == 0)
     irc = MAXLOC(ABS(a), outerand(lpiv, lpiv))
     ipiv(icol) = ipiv(icol) + 1
     IF (ipiv(icol) > 1) CALL nrerror('gaussj: singular matrix(1)')

     ! We now have the pivot element, so we interchange rows, if needed,
     ! to put the pivot element on the diagonal. The columns are not
     ! physically interchanged, only relabeled: indxc(i), the column of
     ! the ith pivot element, is the ith column that is reduced, while
     ! indxr(i) is the row in which that pivot element was originally
     ! located. If indxr(i) \= indxc(i) there is an implied column
     ! interchange. With this form of bookkeeping, the solution b's
     ! will end up in the correct order, and the inverse matrix will be
     ! scrambled by columns.

     IF (irow /= icol) THEN
        CALL swap(a(irow, :), a(icol, :))
        CALL swap(b(irow, :), b(icol, :))
     END IF
     indxr(i) = irow     ! We are now ready to divide the pivot row by the
     indxc(i) = icol     ! pivot element, located at irow and icol.
     IF (a(icol, icol) == 0.0) &
          CALL nrerror('gaussj: singular matrix(2)')
     pivinv=1.0_sp/a(icol, icol)
     a(icol, icol)=1.0
     a(icol, :)=a(icol, :)*pivinv
```

```
    b(icol, :)=b(icol, :)*pivinv
    dumc=a(:, icol)     ! Next we reduce the rows, except for the pivot one.
    a(:, icol)=0.0
    a(icol, icol)=pivinv
    a(1:icol-1,:)=a(1:icol-1,:) - outerprod(dumc(1:icol-1), a(icol,:))
    b(1:icol-1,:)=b(1:icol-1,:) - outerprod(dumc(1:icol-1), b(icol,:))
    a(icol+1:,:)=a(icol+1:,:) - outerprod(dumc(icol+1:), a(icol,:))
    b(icol+1:,:)=b(icol+1:,:) - outerprod(dumc(icol+1:), b(icol,:))
  END DO

  ! It only remains to unscramble the solution in view of the column
  ! interchanges. We do this by interchanging pairs of columns in the
  ! revers order that the permutation was built up.

  DO l = n, 1, -1
    CALL swap(a(:, indxr(l)), a(:, indxc(l)))
  END DO

END SUBROUTINE gaussj
```

5.1.2 The Discussion

The subroutine is implemented in the form of a large `DO` loop that moves down the diagonal while performing the operations of the Gauss Jordan elimination on the resulting submatrix, i.e., on all that between the current point and the bottom right corner of the matrix.

The first step is to locate the pivot element. This is done by calling Fortran intrinsic `MAXLOC`:

```
    irc = MAXLOC(ABS(a), outerand(lpiv, lpiv))
```

The location of the pivot is placed in the array `irc`, whose entries are pointed to by `irow` and `icol`.

Next we swap rows pointed to by `irow` and `icol`, so that the pivot ends up on the diagonal in the `(icol, icol)` location. The original placement of the pivot term is memorised on `indxr` and `indxc` for unscrambling at the end of the procedure:

```
    indxr(i) = irow     ! We are now ready to divid the pivot row by the
    indxc(i) = icol     ! pivot element, located at irow and icol.
```

If that term, i.e., the pivot term happens to be 0 then the matrix, obviously, is singular, so we must abort raising an error flag:

```
    IF (a(icol, icol) == 0.0) &
         CALL nrerror('gaussj: singular matrix(2)')
```

Now we perform the first division of the Gauss-Jordan elimination procedure, i.e., we devide by A_{kk}:

```
    pivinv=1.0_sp/a(icol, icol)
    a(icol, icol)=1.0
    a(icol, :)=a(icol, :)*pivinv
    b(icol, :)=b(icol, :)*pivinv
```

and then reduce all other rows of the matrix. Observe that the whole column `a(:, icol)` is memorized first on `dumc`. We will need it, because, in the meantime we're setting that column of $\boldsymbol{A}$ to

$$\begin{pmatrix} 0 \\ 0 \\ \vdots \\ 1/A_{kl} \\ \vdots \\ 0 \end{pmatrix}$$

where $1/A_{kl}$ is the inverse of the pivot.

```
    dumc=a(:, icol)     ! Next we reduce the rows, except for the pivot one.
    a(:, icol)=0.0
    a(icol, icol)=pivinv
```

Why do we do that? That's because we no longer need that column in its previous form, and instead we're now setting it to what the corresponding column of an identity matrix would have become if it was subject to the Gauss-Jordan operations performed so far. This way, as we keep using $\boldsymbol{A}$ we're simultaneously building its inverse in the same space!

Finally we perform the reductions on `b` and at the same time we build the inverse on `a` thusly:

```
    a(1:icol-1,:)=a(1:icol-1,:) - outerprod(dumc(1:icol-1), a(icol,:))
    b(1:icol-1,:)=b(1:icol-1,:) - outerprod(dumc(1:icol-1), b(icol,:))
    a(icol+1:,:)=a(icol+1:,:) - outerprod(dumc(icol+1:), a(icol,:))
    b(icol+1:,:)=b(icol+1:,:) - outerprod(dumc(icol+1:), b(icol,:))
```

The pivoting would have scrambled the matrix, which now needs to be set back in the right order. But we have memorized the sequence of scrambling on `indxr` and `indxc`, so now, we can undo it:

```
DO l = n, 1, -1
   CALL swap(a(:, indxr(l)), a(:, indxc(l)))
END DO
```

The safety check right at the beginning:

```
    ipiv(icol) = ipiv(icol) + 1
    IF (ipiv(icol) > 1) CALL nrerror('gaussj: singular matrix(1)')
```

ensures that we do not hit the same pivot row twice. If that is to happen the matrix must be singular and in that case we abort flagging an error message.

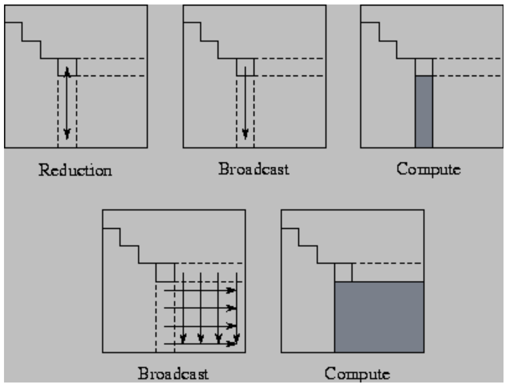

Figure 5.1: Communication and computation in the various phases of the HPF Gaussian (from Foster)

5.1.3 Gauss-Jordan and HPF

The algorithm sweeps down the matrix from the top left corner to the bottom right corner, leaving zero subdiagonal elements behind it.

What is parallel in the algorithm?

1. `MAXLOC`: reduction operation on the row and column defined by the mask `lpiv`, then broadcast within that row and column
2. scale factors require $N - n$ independent operations within column `icol`
3. scale factor and a pivot row value must be broadcast within each column and row respectively
4. the reductions require $\mathcal{O}((N - n)^2)$ independent operations

Attributes of the computation:

- There is little locality in communication, apart from broadcasts and reductions in rows and columns.
- Computation is clustered: much of it is performed in a single row and column, and then, once we get to the reductions, in the bottom right hand corner.
- A `BLOCK` distribution is not going to be advantageous: it would result in many processors being idle!
- Suggested distribution for a small number of processors:

```
!HPF$ ALIGN B(:,:) WITH A(:,:)
!HPF$ DISTRIBUTE A(*,CYCLIC)
```

- If a large number of processors is available:

```
!HPF$ ALIGN B(:,:) WITH A(:,:)
!HPF$ DISTRIBUTE A(CYCLIC,CYCLIC)
```

5.2 LU Decomposition

5.3 Matrix Determinants

5.4 Singular Value Decomposition

5.5 Sparse Linear Systems

Chapter 6

Quantum Computing

6.1 Analog Computers

6.2 Computers as Physical Systems

6.3 Quantum Turing Machines

6.4 Bits and Qubits

6.5 Extracting Answers from Quantum Computers

6.6 Benioff's, Feynman's, and Deutsch's Quantum Computers

6.7 Breaking Unbreakable Codes

6.8 Quantum Cryptography

6.9 Quantum Teleportation

6.10 Quantum Error Correction

6.11 Quantum Computers

6.11.1 Heteropolymer Based Systems

6.11.2 Ion Trap Based Systems

6.11.3 Cavity QED Based Systems

6.11.4 Nuclear Magnetic Resonance Based Systems

Chapter 7

Working with LoadLeveler

7.1 Introduction

LoadLeveler is a job management system. The idea is that given a network of CPU and storage resources, you should simply be able to specify a job you'd like to run, and leave it to LoadLeveler to find the appropriate resources to run the job on, and then to execute and supervise the job, while you can lay back, twitch your thumbs and watch the blinkenlights.

It is not only for your convenience, as a user, that job management systems are deployed at supercomputer centres and often even in smaller computing laboratories or departments. Job management systems ensure that resources are used optimally and that users don't step on each other's toes.

LoadLeveler serves as a job scheduler and provides additional facilities for building, submitting, and processing jobs in a distributed computing environment. It runs on the SP, but can be installed on nearly all desktop machines, MPI farms, even on mainframes. It interacts with NQS (Network Queuing System), and supports *submit-only* machines too.

LoadLeveler documentation can be found on the

> OVPIT AFS cell WWW page

or in

```
/afs/ovpit.indiana.edu/common/www/htdocs/gustav/SP-docs/LoadL
```

On the SP itself you'll find it in

```
/usr/lpp/LoadL/postscript
```

See also

- "Load Leveler Crib"
- "Load Leveler Hints and Recipes"
- "How to Time, Save, and Resubmit LoadLeveler Jobs"

7.2 LoadLeveler Configuration

Before you can begin working with LoadLeveler on the SP you must first find how it is configured. And before you can understand how LoadLeveler has been configured on the SP, you must find how the SP itself is configured. This you can accomplish by typing:

```
gustav@sp20:../gustav 17:35:47 !509 $ jm_status -P
Pool 0:    Batch_only_SP_nodes
  Subpool: BATCH
    Node:  sp01.ucs.indiana.edu
    Node:  sp02.ucs.indiana.edu
    Node:  sp03.ucs.indiana.edu
    ...
    Node:  sp45.ucs.indiana.edu
    Node:  sp46.ucs.indiana.edu
    Node:  sp47.ucs.indiana.edu
gustav@sp20:../gustav 17:35:51 !510 $
```

This command interrogates the SP Job Manager, or Resource Manager, as it is also called. The `-P` option lists *pools of processors* configured into the system. In our case there is just one pool which comprises 47 P2SC nodes. Since every one of those delivers some 700 MFLOPS peak, you've got nearly 33 GFLOPS of computing power available.

Now, once you know what's out there, you can ask LoadLeveler how those resources can be accessed. The command that will tell you that is

```
gustav@sp20:../gustav 17:36:28 !511 $ llclass
Name          MaxJobCPU     MaxProcCPU  Free   Max Description
             d+hh:mm:ss     d+hh:mm:ss Slots Slots

b                    -1             -1    15    24 long serial jobs
l                    -1             -1     5     5 large-memory serial jobs
qcd                  -1             -1     1     1 Quantum Chemistry Division
test         0+00:05:00     0+00:05:00     8     8 5-minute test jobs
q            0+01:00:00     0+01:00:00     2     2 quick serial jobs
a            1+00:00:00     1+00:00:00     4     6 short serial jobs
stat         1+12:00:00     1+12:00:00     3     3 statistics jobs
pa           1+12:00:00     1+12:00:00    12    12 short parallel jobs
math         1+12:00:00     1+12:00:00     3     3 mathematics jobs
pb                   -1             -1     0    32 long parallel jobs
gustav@sp20:../gustav 17:54:17 !512 $
```

This time LoadLeveler tells us that we have 10 classes. LoadLeveler classes correspond closely to *queues* in systems such as NQS and, indeed, there is a queue associated with every class.

Class `pb` has up to 32 slots, of which, according to the listing, none are available at present. Those 32 slots are 32 job instances. That is the class allows you to run either up to 32 serial jobs, or, say, 2 parallel jobs, each running on 16 processors.

Classes `b`, `l`, `qcd`, and `pb` are CPU-time unlimited. This means that you can submit, for example, a 32-way parallel job to class `pb` that may run forever. This may be rather antisocial, but LoadLeveler configuration allows you to do just that.

Class `test` is for test runs only, i.e., for very short jobs, just long enough to check that your program has been correctly linked and that it runs. Then we have classes `q` through `math`, which are for jobs that take between 1 day and 1.5 days of CPU time.

In order to find more information about any particular class, you can call `llclass` with the `-l` switch, e.g.:

```
 gustav@sp20:../gustav 17:54:17 !512 $ llclass -l pa
=============== Class pa ==========
                Name: pa
            priority: 40
               admin:
           NQS_class: F
          NQS_submit:
           NQS_query:
      max_processors: 8
             maxjobs: -1
       class_comment: short parallel jobs
    wall_clock_limit: -1, -1
       job_cpu_limit:   1+12:00:00, -1
           cpu_limit:   1+12:00:00, -1
          data_limit: -1, -1
          core_limit: -1, -1
          file_limit: -1, -1
         stack_limit: -1, -1
           rss_limit: -1, -1
                nice: 0
                free: 12
             maximum: 12
gustav@sp20:../gustav 18:19:58 !513 $
```

Here you can see that even though there are 12 slots in this class, a maximum number of processors you can request is 8. The CPU limit is cumulative, i.e., if you run a job on 8 CPUs and if they all munch CPU time equally, the CPU time allowance per processor will be 4 hours and 30 minutes.

If you run `llclass -l` on the `test` class, you'll see that it has a higher priority than the `pa` class. They both run on the same processors, actually, so if there are two jobs submitted at the same time, one to `pa` and the other one to `test`, it is the `test` jobs that will run first – unless users alter the priorities of those jobs explicitely. A user can do that, but user priority has a smaller weight usually than a system priority.

How to find out which class runs on which nodes? To do that you can run the command `llstatus`:

```
gustav@sp20:../SP 18:32:22 !544 $ llstatus
Name                      Schedd  InQ Act Startd Run LdAvg Idle Arch      OpSys
libra.ucs.indiana.edu     Avail     2   2 Idle     0 0.09  2112 R6000     AIX43
sp01.ucs.indiana.edu      Avail    36   8 Run      1 1.10     1 R6000     AIX43
sp02.ucs.indiana.edu      Avail     5   4 Run      1 1.02  7801 R6000     AIX43
sp03.ucs.indiana.edu      Avail     0   0 Run      1 1.08  2952 R6000     AIX43
...
sp44.ucs.indiana.edu      Avail     0   0 Run      1 1.00  9112 R6000     AIX43
sp45.ucs.indiana.edu      Avail     0   0 Run      1 1.00  9999 R6000     AIX43
sp46.ucs.indiana.edu      Avail     0   0 Busy     2 2.03  9999 R6000     AIX43
```

```
sp47.ucs.indiana.edu      Avail     0    0 Run      1 1.04  9999 R6000     AIX43

R6000/AIX43           48 machines  44 jobs  43 running
Total Machines        48 machines  44 jobs  43 running

The Central Manager is defined on sp01.ucs.indiana.edu

All machines on the machine_list are present
gustav@sp20:../SP 18:32:24 !545 $
```

When called without any options, `llstatus` simply lists all machines under the LoadLeveler management. Observe that although our SP has 47 nodes, LoadLeveler manages 48 machines. The 48th machine is `libra.ucs.indiana.edu`. The listing tells you if a machine is busy or idle, what is the average load on the machine, what is its architecture, operating system, and whether the LoadLeveler scheduler runs on that node.

When invoked with the `-l` option, the command `llstatus` returns a very detailed listing for each machine that is managed by LoadLeveler. If you don't want to look at all nodes, you can just select one providing its name on the command line:

```
gustav@sp20:../SP 18:38:13 !551 $ llstatus -l sp20
name: "sp20.ucs.indiana.edu"
machine_context:
Running            = 0
ScheddAvail        = 1
StartdAvail        = 1
State              = Idle
ScheddState        = 0
OpSys              = AIX43
Arch               = R6000
Machine            = sp20.ucs.indiana.edu
START              = T
SUSPEND            = F
CONTINUE           = T
VACATE             = F
KILL               = F
SYSPRIO            = ((ClassSysprio *  100) -  QDate)
MACHPRIO           = (0 -  (1000 *  (LoadAvg /  Speed)))
VirtualMemory      = 105392
EnteredCurrentState = Tue Jan  5 12:17:37 1999
Disk               = 13072
Tmp                = 197736
KeyboardIdle       = 42
LoadAvg            = 0.000092
AvailableClasses   = { "pa" "test" }
DrainingClasses    = { }
DrainedClasses     = { }
Pool               = 0
Adapter            = { "ethernet" "hps_user" "hps_ip" }
ConfiguredClasses  = { "pa" "test" }
Feature            = { "256MB" "afs" }
ProtocolVersion    = 1
CkptVersion        = 1
Memory             = 256
Max_Starters       = 2
ConfigTimeStamp    = Tue Jan  5 12:16:32 1999
Cpus               = 1
Speed              = 3.000000
MasterMachPriority  = 0.000000
Subnet             = 129.79.7
CustomMetric       = 1
```

```
ScheddRunning       = 0
Pending             = 0
Starting            = 0
Idle                = 0
Unexpanded          = 0
Held                = 0
Removed             = 0
RemovePending       = 0
Completed           = 2
DependantNotRun     = 0
TotalJobs           = 0
time_stamp: Tue Jan 12 18:37:42 1999

gustav@sp20:../SP 18:38:19 !552 $
```

There is quite a lot of information in this listing. In particular you'll see the entry `ConfiguredClasses`, which in this case is: `{ "pa" "test" }`, and this means that when you submit a job to `pa` or to `test` it may end up running on that node. Or on some other node that has `pa` or `test` in its `ConfiguredClasses` slot.

It would be good, however, if we could ask LoadLeveler about a particular class and then find which nodes it runs on. The command `llclass` should do that, but it doesn't. So on our system we have our own local command, which is `llconfig` and that command prints a more palatable summary:

```
gustav@sp20:../SP 18:54:26 !556 $ llconfig

       LoadLeveler Configuration on the SP


                               Total
   Node       Job Classes      Jobs  Features

  libra         q                2   512MB
  sp01          l,b              2   512MB
  sp02          l,b              2   512MB
  sp03          l,pb             2   512MB
  sp04          l,pb             2   512MB
  sp05          stat,pb          2   256MB  gauss glim lisrel prelis rats sas spss tsp
  sp06          stat,pb          2   256MB  gauss glim rats sas spss tsp
  sp07          stat,pb          2   256MB  glim rats sas spss tsp
  sp08          b,pb             2   256MB
  sp09          math,pb          2   256MB  lindo lingo maple math matlab
  sp10          math,pb          2   256MB  lindo lingo maple matlab
  sp11          math,pb          2   256MB  lindo lingo maple matlab
  sp12          b,pb             2   256MB
  sp13          b,pb             2   256MB
  sp14          b,pb             2   256MB
  sp15          b,pb             2   256MB  naglib
  sp16          b,pb             2   256MB  naglib
  sp17          pa,test          2   256MB  afs
  sp18          pa,test          2   256MB  afs
  sp19          pa,test          2   256MB  afs
  sp20          pa,test          2   256MB  afs
  sp21          pa,test          2   256MB  afs
  sp22          pa,test          2   256MB  afs
  sp23          pa,test          2   256MB  afs
  sp24          pa,test          2   256MB  afs
  sp25          l,qcd            2   512MB
  sp26          b,pb             2   256MB  bigscr naglib
  sp27          b,pb             2   256MB  bigscr naglib
  sp28          b,pb             2   256MB  bigscr naglib
  sp29          b,pb             2   256MB  bigscr naglib
  sp30          b,pb             2   256MB  bigscr naglib
  sp31          b,pb             2   256MB  bigscr naglib
  sp32         b,pb              2   256MB  bigscr naglib
  sp33         b,pb              2   256MB  bigscr naglib
  sp34         b,pb              2   256MB  bigscr naglib
  sp35         b,pb              2   256MB  bigscr naglib
  sp36         b,pb              2   256MB  bigscr naglib
  sp37         b,pb              2   256MB  bigscr naglib
  sp38         b,pb              2   256MB  bigscr naglib
  sp39         b,pb              2   256MB  bigscr naglib
  sp40         a,pa              2   256MB  afs bigscr
  sp41         a,pa              2   256MB  afs bigscr
  sp42         a,pa              2   256MB  afs bigscr
  sp43         a,pa              2   256MB  afs bigscr
  sp44         a,pb              2   256MB  bigscr
  sp45         a,pb              2   256MB
  sp46         b,pb              2   256MB  bigscr
  sp47         b,pb              2   256MB  bigscr

  Maximum Processor Limits

  class pa             8
  class test           8
  class pb            32
  all other classes    1

  Memory

  42 nodes have 256MB memory.  The 6 nodes with 512MB memory can
  be selected by feature code, providing the appropriate class is
  also specified.
gustav@sp20:../SP 18:55:16 !557 $
```

To search for a more specific information you can always `grep`, for example:

```
gustav@sp20:../SP 18:55:16 !557 $ llconfig | grep pa
  sp17         pa,test           2   256MB  afs
  sp18         pa,test           2   256MB  afs
  sp19         pa,test           2   256MB  afs
  sp20         pa,test           2   256MB  afs
  sp21         pa,test           2   256MB  afs
  sp22         pa,test           2   256MB  afs
  sp23         pa,test           2   256MB  afs
  sp24         pa,test           2   256MB  afs
  sp40         a,pa              2   256MB  afs bigscr
  sp41         a,pa              2   256MB  afs bigscr
  sp42         a,pa              2   256MB  afs bigscr
  sp43         a,pa              2   256MB  afs bigscr
  class pa             8
gustav@sp20:../SP 18:56:19 !558 $
```

And this clearly tells us that class `pa` runs on `sp17` through `sp24` and then on `sp40` through `sp43`. The listing also tells us that all those nodes run AFS. They are often used for Computer Science experiments, and you can expect to find various other goodies installed there soon, e.g., DFS, HPSS, and GPFS.

7.3 Submitting, Inspecting, and Cancelling LoadLeveler Jobs

So, how does one pass a job on to LoadLeveler for execution, and having passed, how does one check what's going on with that job, and having checked and changed one's mind, how does one cancel the job?

One prepares a job by creating a job description file. A LoadLeveler job description file comprises a number of LoadLeveler directives, and possibly also a shell or a Perl script that follows the directives.

LoadLeveler directives are a little like High Performance Fortran directives. To a shell or to Perl that may be invoked by LoadLeveler to interpret the script they look like comments, so they ignore them. But LoadLeveler reads the directives and performs various additional actions as instructed.

Here is an example of a LoadLeveler job description file:

```
gustav@sp20:../LoadLeveler 20:06:15 !577 $ cat echo.ll
#@ output      = echo.out
#@ error       = echo.err
#@ class       = test
#@ environment = COPY_ALL
#@ executable  = /afs/ovpit.indiana.edu/@sys/gnu/bin/echo
#@ arguments   = hello world
#@ queue
gustav@sp20:../LoadLeveler 20:06:17 !578 $
```

LoadLeveler directives begin with `#@`, which to all IEEE-1003.2 compliant shells and to Perl looks like this is a comment. Unfortunately neither Common Lisp nor Scheme interpret `#` as a comment character. It would be nice if LoadLeveler directive flag could be changed. The LoadLeveler directive flag, `#@`, must be followed by a keyword, such as `output` or `class`, and this, in turn, may be followed by additional parameters, if that is required by the keyword.

In simplest situations you would specify an executable to be run by LoadLeveler by something like:

```
#@ executable  = /afs/ovpit.indiana.edu/@sys/gnu/bin/echo
```

and if additional command line arguments need to be used, you would specify them by something like:

```
#@ arguments   = hello world
```

The job itself is queued by the directive:

```
#@ queue
```

This keyword *must* appear in the LoadLeveler job description file at least once.

Having prepared the job description file, submit it with the command `llsubmit`:

```
gustav@sp20:../LoadLeveler 20:10:16 !583 $ ls
echo.ll
gustav@sp20:../LoadLeveler 20:18:48 !584 $ llsubmit echo.ll
submit: The job "sp20.26" has been submitted.
gustav@sp20:../LoadLeveler 20:18:54 !585 $ ls
echo.ll   echo.out
gustav@sp20:../LoadLeveler 20:18:59 !586 $ cat echo.out
hello world
gustav@sp20:../LoadLeveler 20:19:02 !587 $
```

Assuming that the job executed correctly and without errors or diagnostics, the output will be left on whatever file you have specified with the

```
#@ output      = echo.out
```

directive.

In this case the job has been sent to class `test` following the directive:

```
#@ class       = test
```

The job may not run for a while depending on what other jobs there are in the system, queue priorities, job priorities, etc. In general you cannot predict which node exactly will the job run on. In this case it may run on any node that supports the `test` class.

When the job completes, you will find file `echo.out` in your working directory and inside the file the two magic words:

```
gustav@sp20:../LoadLeveler 20:20:57 !588 $ cat echo.out
hello world
gustav@sp20:../LoadLeveler 20:23:20 !589 $
```

You can accomplish a similar result by submitting the following LoadLeveler job description file:

```
gustav@sp20:../LoadLeveler 20:26:29 !600 $ cat echo-2.ll
#@ output      = echo-2.out
#@ error       = echo-2.err
#@ class       = test
#@ environment = COPY_ALL
#@ shell       = /afs/ovpit.indiana.edu/@sys/gnu/bin/bash
#@ queue
echo hello world
gustav@sp20:../LoadLeveler 20:26:33 !601 $
```

This time we tell LoadLeveler that the job description file is a shell script and that it should invoke

```
/afs/ovpit.indiana.edu/@sys/gnu/bin/bash
```

to interpret it. The script itself contains just:

```
echo hello world
```

and, indeed, when the job completes, you will find a file `echo.out` in your working directory with the following words in it:

```
gustav@sp20:../LoadLeveler 20:26:33 !601 $ cat echo-2.out
hello world
gustav@sp20:../LoadLeveler 20:28:28 !602 $
```

There is a significant difference between the two runs. In the first case we run the stand-alone `echo` binary from

```
/afs/ovpit.indiana.edu/@sys/gnu/bin
```

In the second case the binary is not `echo` but `bash`, and `bash` built-in `echo` is used to print `hello world` on standard output.

This job is way to short to capture it on the queue, unless there are no spare slots left and the job has to wait. But assuming that you'd have a long job or that the queue would be fully occupied, how would you check what is happening to your job?

The command is `llq`. Here's an example of how it works:

```
gustav@sp20:../LoadLeveler 20:30:48 !603 $ llq
Id                       Owner      Submitted   ST PRI Class        Running On
------------------------ ---------- ----------- -- --- ------------ -----------
sp01.71.0                rshoward    1/9  08:04 R  50  b            sp01
sp01.82.0                rshoward    1/10 08:27 R  50  b            sp02
sp01.5.22                wfischer    1/2  08:03 R  50  pb           sp06
sp01.5.21                wfischer    1/2  08:03 R  50  pb           sp11
sp02.1974.0              eisenste    1/11 03:57 R  50  b            sp13
sp05.1150.0              kang        1/8  16:06 R  50  b            sp27
libra.1849.0             kapihaka    1/10 16:20 R  50  b            sp28
libra.1838.0             tachim      1/5  16:48 R  50  b            sp32
sp02.2000.0              eisenste    1/12 03:24 R  50  b            sp33
sp01.5.20                wfischer    1/2  08:03 R  50  pb           sp34
sp01.5.19                wfischer    1/2  08:03 R  50  pb           sp35
sp02.1953.0              eisenste    1/8  02:57 R  50  b            sp36
sp01.134.0               tghanty     1/12 20:27 R  50  a            sp40
sp01.128.0               tghanty     1/12 12:56 R  50  a            sp41
sp01.133.0               tghanty     1/12 19:48 R  50  a            sp43
sp02.1955.0              eisenste    1/8  03:01 R  50  b            sp46
sp02.2001.0              eisenste    1/12 03:26 NQ 50  b
sp01.5.23                wfischer    1/2  08:03 NQ 50  pb
sp01.5.24                wfischer    1/2  08:03 NQ 50  pb
sp01.5.25                wfischer    1/2  08:03 NQ 50  pb
sp01.5.26                wfischer    1/2  08:03 NQ 50  pb
sp01.5.27                wfischer    1/2  08:03 NQ 50  pb
sp01.69.0                wfischer    1/8  21:41 NQ 50  pb
sp01.69.1                wfischer    1/8  21:41 NQ 50  pb
sp01.69.2                wfischer    1/8  21:41 NQ 50  pb
sp01.69.3                wfischer    1/8  21:41 NQ 50  pb
sp01.5.0                 wfischer    1/2  08:03 C  50  pb
sp01.5.1                 wfischer    1/2  08:03 C  50  pb
sp01.5.2                 wfischer    1/2  08:03 C  50  pb
sp01.5.3                 wfischer    1/2  08:03 C  50  pb
sp01.5.4                 wfischer    1/2  08:03 C  50  pb
sp01.5.5                 wfischer    1/2  08:03 RM 50  pb
sp01.5.6                 wfischer    1/2  08:03 C  50  pb
sp01.5.7                 wfischer    1/2  08:03 C  50  pb
sp01.5.8                 wfischer    1/2  08:03 C  50  pb
sp01.5.9                 wfischer    1/2  08:03 C  50  pb
sp01.5.10                wfischer    1/2  08:03 C  50  pb
sp01.5.11                wfischer    1/2  08:03 C  50  pb
sp01.5.12                wfischer    1/2  08:03 C  50  pb
```

```
sp01.5.13                wfischer    1/2  08:03 C  50  pb
sp01.5.14                wfischer    1/2  08:03 C  50  pb
sp01.5.15                wfischer    1/2  08:03 C  50  pb
sp01.5.16                wfischer    1/2  08:03 C  50  pb
sp01.5.17                wfischer    1/2  08:03 C  50  pb
sp01.5.18                wfischer    1/2  08:03 C  50  pb

26 jobs in queue 0 waiting, 0 pending, 16 running, 10 held.
gustav@sp20:../LoadLeveler 20:30:58 !604 $
```

This listing tells us a lot of things. For example that Mr Will Fischer is hogging the system and that Mary Papakhian should ever so gently infuse some sanity into him. It also tells us that there is little point submitting any jobs to the `pb` class, because the queue is clogged with Mr Fischer's jobs.

The command `llq` supports various options. For example, to list only jobs in class `b` type

```
gustav@sp20:../LoadLeveler 20:40:37 !614 $ llq -c b
Id                       Owner      Submitted   ST PRI Class        Running On
------------------------ ---------- ----------- -- --- ------------ -----------
sp01.71.0                rshoward    1/9  08:04 R  50  b            sp01
sp01.82.0                rshoward    1/10 08:27 R  50  b            sp02
sp02.1974.0              eisenste    1/11 03:57 R  50  b            sp13
sp05.1150.0              kang        1/8  16:06 R  50  b            sp27
libra.1849.0             kapihaka    1/10 16:20 R  50  b            sp28
libra.1838.0             tachim      1/5  16:48 R  50  b            sp32
sp02.2000.0              eisenste    1/12 03:24 R  50  b            sp33
sp02.1953.0              eisenste    1/8  02:57 R  50  b            sp36
sp02.1955.0              eisenste    1/8  03:01 R  50  b            sp46
sp02.2001.0              eisenste    1/12 03:26 NQ 50  b

10 jobs in queue 0 waiting, 0 pending, 9 running, 1 held.
gustav@sp20:../LoadLeveler 20:40:59 !615 $
```

The listing tells us who and when submitted the jobs, what the jobs' priority is, what they run on, when they finally do, and which class they've been submitted to. Also, what is the job ID, and what it the job's current status. The status is one of the following:

C The job has completed.

D The job has been deferred.

H A user put a hold on the job.

I No machine has been selected for the job yet. The job is idle.

NQ The job is not currently considered to run on any machine. It has not been queued.

NR The job is badly formulated: there are unresolvable dependencies on other jobs in it, so it will never run.

P The job is in the process of starting on one or more machines: it is pending.

R The job is running.

RM The job was removed (cancelled) either by the user or by LoadLeveler.

RP The job is in the process of being removed: not all machines have responded yet, so the removal is pending.

S The LoadLeveler administrator, i.e., Mary Papakhian, has put the job on hold. This is called *system hold*.

SH Both the LoadLeveler administrator and the user have put the job on hold.

ST The job has been dispatched and received by a target machine. LoadLeveler is setting up the environment for the job. The job is said to be *starting*.

V The job did not complete for some reason. It has been *vacated*.

How does one put a hold on a job? One issues the command `llhold` giving it the job ID as an argument:

```
gustav@sp20:../LoadLeveler 20:52:18 !626 $ llhold sp01.5.23
hold: Hold command has been sent to central manager on "sp01.ucs.indiana.edu"
gustav@sp20:../LoadLeveler 20:52:34 !627 $
```

It's not going to work here, because it's not my job.
To release a job from hold type:

```
gustav@sp20:../LoadLeveler 20:54:38 !635 $ llhold -r sp01.5.23
hold: Hold command has been sent to central manager on "sp01.ucs.indiana.edu"
gustav@sp20:../LoadLeveler 20:54:54 !636 $
```

It may happen that you want to cancel the job altogether. The command to do that is `llcancel`, e.g.,

```
gustav@sp20:../LoadLeveler 20:55:42 !639 $ llcancel sp01.5.20
llcancel: Cancel command has been sent to central manager on "sp01.ucs.indiana.edu"
gustav@sp20:../LoadLeveler 20:56:23 !640 $
```

7.4 Specification of LoadLeveler Jobs

7.4.1 Interactive LoadLeveler Jobs

Contrary to what it may appear at first glance, it is quite easy to run interactive and graphic jobs under LoadLeveler. Of course you must have a workstation with X11 server and X11 authority running for that to work, but that goes without saying.
On your local workstation issue the command:

```
gustav@blanc:../Notes 17:51:08 !509 $ xauth list
blanc.ovpit.indiana.edu:0  MIT-MAGIC-COOKIE-1  35704e716f6e69723965725843564a54
...
gustav@blanc:../Notes 21:12:17 !510 $
```

Select the whole first line of the listing and switch to an `xterm` running on the SP. There type:

```
gustav@sp20:../LoadLeveler 20:58:34 !643 $ xauth add blanc.ovpit.indiana.edu:0 \
MIT-MAGIC-COOKIE-1  35704e716f6e69723965725843564a54
```

Now you're ready to display X11 applications running on any of the SP nodes on your X11 server. To make sure that it really works type:

```
gustav@sp20:../LoadLeveler 20:59:00 !645 $ xclock -display blanc.ovpit.indiana.edu:0.0 &
```

a clock face should pop up on your workstation's X11 display.

Now prepare the following LoadLeveler job description file:

```
gustav@sp20:../LoadLeveler 21:26:28 !654 $ cat xterm.ll
# @ output = $(job_name).out
# @ error = $(job_name).err
# @ job_type = serial
# @ class = test
# @ notification = always
# @ environment = COPY_ALL
# @ executable = /afs/ovpit.indiana.edu/@sys/X11R6.4/bin/xterm
# @ arguments = -bg white -ls -sb -sl 300 -n $(job_name) -T $(job_name)
# @ queue
gustav@sp20:../LoadLeveler 21:26:32 !655 $
```

Now submit this job with

```
gustav@sp20:../LoadLeveler 21:26:32 !655 $ llsubmit xterm.ll
submit: The job "sp20.29" has been submitted.
gustav@sp20:../LoadLeveler 21:29:50 !656 $
```

and an xterm window entitled `sp20.ucs.indiana.edu.29` will pop up on your X11 display.

Why should one submit interactive jobs to LoadLeveler instead of running them simply by connecting to a node? The answer is that on many SP systems you cannot `telnet` or `rlogin` or `slogin` to a node. There may be a usually overloaded so called *front end* system, often an SMP node, and P2SC nodes are accessible through LoadLeveler only.

In the LoadLeveler job specification above I have used a number of new keywords and LoadLeveler variables.

The keyword `job_type` specifies whether the job is a `serial` job or a `parallel` one. Instead of hardwiring the name of the output and error logs, here I have used a LoadLeveler variable `$(job_name)`, which is going to evaluate to whatever the job name is going to be, e.g., `sp20.ucs.indiana.edu.29`. I have also asked for a `notification` to be sent to me on commencement and on termination of the job. Normally it will be sent on termination only.

You can submit Emacs, Octave, Mathematica, Maple, Lisp, AVS and many other interactive jobs that way too.

csh and tcsh users may encounter some problems trying to pull this trick. Neither shell is IEEE-1003.2 compliant and at times they don't seem to interoperate with LoadLeveler well. My suggestion to csh and tcsh users is to switch to bash. You'll get much the same interactive environment, whereas the shell is fully complian with IEEE-1003.2 specs, and has no problems interacting with LoadLeveler.

7.4.2 Submitting a not quite interactive job

There is a class of jobs, which many users who lack UNIX skills think of as interactive jobs, but, which, in fact, aren't interactive at all. Applications which take input from a command line in some sort of an application dependent language fall in that category. Examples are Common Lisp, Smalltalk, Scheme, Octave, Matlab, Mathematica, Xplor, GeneHunter, etc.

Often a user has a command file prepared, which must be loaded into an application from an interactive session. Once the command file is loaded the application begins executing a program, which may take hours to complete. A user at that stage goes away leaving an active telnet connection or an X11 window on the display. The window was needed only to load the file and perhaps issue some start-up commands for the computation.

Jobs like that are not interactive at all and they can and should be run under LoadLeveler without asking for an xterm window and forking an unnecessary login shell.

The way to execute such jobs is to use the here-input feature of UNIX shells:

```
$ my_command << EOF
   one_line_of_input
   another_line_of_input
EOF
```

Here is an example:

```
# @ shell = /afs/ovpit.indiana.edu/@sys/gnu/bin/bash
# @ output = hello-lisp.out
# @ error = hello-lisp.err
# @ job_type = serial
# @ class = test
# @ notification = always
# @ environment = COPY_ALL
# @ queue
clisp -q << EOF
(load "hello.fas")
(hello)
EOF
```

Here I have a Common Lisp program stored on a file hello.fas. Normally, in order to execute that program I would have to enter a Common Lisp environment with the command clisp. Then I would have to load the file which contains the program, and finally I would have to evaluate the function, which has been defined on hello.fas, by typing (hello).

But all that can be accomplished also by typing

```
clisp -q << EOF
(load "hello.fas")
(hello)
EOF
```

in a shell script without having to invoke xterm first. A UNIX shell (csh, tcsh, sh, ksh or bash) on encountering a construct like that will take the text enclosed by the

```
<< EOF
...
EOF
```

construct and will pass it to the program, in this case `clisp`, as if it has been typed in by the user.

Long Matlab, GeneHunter, Xplor, etc., computations can and should be run like this too.

7.4.3 Submitting a Simple Sequential Batch Job

By a simple batch job I mean running just one program under LoadLeveler, a little like in the interactive example above, without any pre or post-processing. Consequently the LoadLeveler script looks quite similar too.

Here is a simple hello world program written in Emacs Lisp:

```
gustav@sp20:../LoadLeveler 23:26:47 !745 $ cat hello.el
(defun hello ()
   (princ "hello world\n"))
gustav@sp20:../LoadLeveler 23:27:01 !746 $
```

To execute this program in the Emacs batch mode under LoadLeveler I had first saved it on a file `hello.el`. Then I edited the following LoadLeveler script:

```
gustav@sp20:../LoadLeveler 23:27:01 !746 $ cat emacs-batch.ll
# @ executable = /afs/ovpit.indiana.edu/@sys/gnu/bin/emacs
# @ arguments = -batch -l hello.el -f hello
# @ output = emacs-batch.out
# @ error = emacs-batch.err
# @ job_type = serial
# @ class = test
# @ notification = always
# @ environment = COPY_ALL
# @ queue
gustav@sp20:../LoadLeveler 23:27:22 !747 $
```

and saved it on `emacs-batch.ll`.

The script was submitted to LoadLeveler with the command:

```
gustav@sp20:../LoadLeveler 23:27:22 !747 $ llsubmit emacs-batch.ll
submit: The job "sp20.33" has been submitted.
gustav@sp20:../LoadLeveler 23:28:00 !748 $
```

When the job had finished its execution there was a file `emacs-batch.out` left in my working directory:

```
gustav@sp20:../LoadLeveler 23:28:00 !748 $ cat emacs-batch.out
hello world
gustav@sp20:../LoadLeveler 23:28:24 !749 $
$
```

An alternative way is to make the primary LoadLeveler job a shell script, and to execute emacs from within it. In that case you must not use the `#@executable` and the `#@arguments` directives. Instead, use the `#@shell` directive, to specify the shell of choice for your command file. Here is an example:

```
# @ shell = /afs/ovpit.indiana.edu/@sys/gnu/bin/bash
# @ output = emacs-batch.out
# @ error = emacs-batch.err
# @ job_type = serial
# @ class = test
# @ notification = always
# @ environment = COPY_ALL
# @ queue
emacs -batch -l hello.el -f hello
```

7.4.4 Submitting a more complex sequential batch job

Often you may wish to perform some preliminary manipulations on your data files before passing them on to your application for execution, and after that's done, you may wish to do some clean-up work, perhaps making sure that various scratch files have been removed, etc.

The way to do that is to use the second approach presented in the previous section, i.e., to submit a shell script to LoadLeveler.

Below is an example of a job like that. This job comprises three steps:

1. Obtain information about LoadLeveler variables and write them on a data file. The information is obtained by running `env` and `grep` and the data file is constructed by running `awk`. The data file is written in the form of an Emacs Lisp program.

2. Invoke the application on the data file, which has been generated dynamically in the previous step. The application, in this case, is emacs, and the data file is the Emacs Lisp program saved on `llenv.el`.

3. Cleanup, i.e., remove the data file generated in step 1, since it is no longer needed.

Of course, there are simpler ways to list LoadLeveler environmental variables, but this toy example neatly illustrates the idea of a three step procedure:

1. data preprocessing/generation

2. data manipulation by the main application

3. data cleanup

Most simple LoadLeveler jobs have this structure.

```
gustav@sp20:../LoadLeveler 23:23:59 !735 $ cat env.ll
# @ shell = /afs/ovpit.indiana.edu/@sys/gnu/bin/bash
# @ output = env.out
# @ error = env.err
# @ job_type = serial
# @ class = test
# @ notification = always
# @ environment = COPY_ALL
# @ queue
env | grep LOADL | \
awk ' BEGIN {
         { printf "(defun llenv ()\n" }
         { printf "   (princ \"LoadLeveler variables:\\n\") " }
      }
      { printf "   (princ \"\t%s\\n\")\n", $0 }
      END { print ")"} ' > llenv.el
emacs -batch -l llenv.el -f llenv > llenv.out
rm llenv.el
gustav@sp20:../LoadLeveler 23:24:18 !736 $
```

Having saved this LoadLeveler script on `env.ll` I have submitted it with

```
gustav@sp20:../LoadLeveler 23:24:44 !737 $ llsubmit env.ll
submit: The job "sp20.32" has been submitted.
gustav@sp20:../LoadLeveler 23:24:52 !738 $
```

and then viewed the results of the run as follows:

```
gustav@sp20:../LoadLeveler 23:25:18 !739 $ cat llenv.out
LoadLeveler variables:
        LOADL_STEP_GROUP=ucs
        LOADL_JOB_NAME=sp20.ucs.indiana.edu.32
        LOADL_STEP_NAME=0
        LOADL_STEP_CLASS=test
        LOADL_STEP_ID=sp20.ucs.indiana.edu.32.0
        LOADL_STEP_OWNER=gustav
        LOADL_ACTIVE=1.3.0.18
        LOADL_STEP_ARGS=
        LOADL_STEP_IN=/dev/null
        LOADLBATCH=yes
        LOADL_STEP_ERR=env.err
        LOADL_STEP_ACCT=
        LOADL_STEP_COMMAND=env.ll
        LOADL_PROCESSOR_LIST=sp17.ucs.indiana.edu
        LOADL_STARTD_PORT=9611
        LOADL_STEP_NICE=0
        LOADL_STEP_OUT=env.out
        LOADL_STEP_TYPE=SERIAL
        LOADL_STEP_INITDIR=/N/u/gustav/SP/LoadLeveler
gustav@sp20:../LoadLeveler 23:25:22 !740 $
```

7.4.5 Submitting a Number of Dependent Jobs

The postprocessing or preprocessing of data may sometimes be so involved that it should be performed as a separate LoadLeveler job, rather than combined with the main computational task.

The simplest way to procede in such situation would be to submit one LoadLeveler job, then wait for it to finish execution, and then to submit the second job. The submission of the second job could be performed from within the LoadLeveler script of the first job.

The following two scripts split the example from Section 7.4.4 into two steps.

The first script, called `env-1.ll` uses commands `env`, `grep`, and `awk` to generate a data file, in this case an Emacs Lisp code, which is saved on `llenv.el` (remember that programs are data, and, in particular, in case of Lisp, there is no semantic difference between programs and data: both are stored in the same data section of a Lisp process, and both can be modified dynamically during program execution). Once `awk` exists the script checks if the data file is there (for example, an error may have occurred while executing `awk`). It also checks if the second LoadLeveler script can be found in its working directory. If both files are present, the second script is submitted with the `llsubmit` command.

```
gustav@sp20:../LoadLeveler 23:40:22 !751 $ cat env-1.ll
# @ shell = /afs/ovpit.indiana.edu/@sys/gnu/bin/bash
# @ output = env-1.out
# @ error = env-1.err
# @ job_type = serial
# @ class = test
# @ notification = always
# @ environment = COPY_ALL
# @ queue
env | grep LOADL | \
awk ' BEGIN {
         { printf "(defun llenv ()\n" }
         { printf "   (princ \"LoadLeveler variables:\\n\") " }
      }
      { printf "   (princ \"\t%s\\n\")\n", $0 }
      END { print ")"} ' > llenv.el
if [ -f llenv.el -a -f env-2.ll ]
then
   llsubmit env-2.ll
fi
gustav@sp20:../LoadLeveler 23:40:25 !752 $
```

The second script is called `env-2.ll`. First it checks if the file `llenv.el` exists. Even though we have already checked that within `env-1.ll`, here we do so again, because the scripts are separate, and there is always a possibility that `env-2.ll` may have been submitted without running `env-1.ll` first. If the file exists, we run emacs on it, if it doesn't, we flag an error and exit. The data file itself, `llenv.el`, is removed after emacs had its way with it.

```
gustav@sp20:../LoadLeveler 23:43:10 !758 $ cat env-2.ll
# @ shell = /afs/ovpit.indiana.edu/@sys/gnu/bin/bash
# @ output = env-2.out
# @ error = env-2.err
# @ job_type = serial
# @ class = test
# @ notification = always
# @ environment = COPY_ALL
# @ queue
if [ -f llenv.el ]
then
   emacs -batch -l llenv.el -f llenv > llenv.out
else
```

```
   echo Error: env-2.ll job: llenv.el not found
   exit 1
fi
rm llenv.el
gustav@sp20:../LoadLeveler 23:43:27 !759 $
```

It is easy to restructure the two scripts above into one script, which first performs one task, then resubmits itself, and on the second invocation performs the second task.

In order to do that, the script must be able to find out on its own, whether its current instantiation is the first or the second one. If you have a creepy feeling now that we are getting close to talking about reincarnation, well, yes, you're quite right. That's exactly what we're talking about! How can a process know that it already lived before?

The answer is: by inspecting its environment and finding a particular variable set. The variable would be set during the first instantiation of the LoadLeveler job. It would not exist at all outside of those LoadLeveler jobs, i.e., the user should make sure that it is unset in the user's normal environment.

Here's the script:

```
gustav@sp20:../LoadLeveler 23:51:33 !803 $ cat env-3.ll
# @ shell = /afs/ovpit.indiana.edu/@sys/gnu/bin/bash
# @ output = $(job_name).out
# @ error = $(job_name).err
# @ job_type = serial
# @ class = test
# @ notification = always
# @ environment = COPY_ALL
# @ queue
if [ -z "$ENV_SECOND_SUBMISSION" ]
then
   env | grep LOADL | \
   awk ' BEGIN {
            { printf "(defun llenv ()\n" }
            { printf "   (princ \"LoadLeveler variables:\\n\") " }
         }
         { printf "   (princ \"\t%s\\n\")\n", $0 }
         END { print ")"} ' > llenv.el
   if [ $? -eq 0 ]
   then
      export ENV_SECOND_SUBMISSION="yes"
      llsubmit $LOADL_STEP_COMMAND
   else
      echo Error: problem executing awk
      exit 1
   fi
else
   emacs -batch -l llenv.el -f llenv > llenv.out
   rm llenv.el
fi
gustav@sp20:../LoadLeveler 23:52:16 !804 $
```

The script works as follows. The first step is to check if the environmental variable `ENV_SECOND_SUBMISSION` has been set to something. If not, it means that this instantiation of the job has no ancestor. In this case the script calls `env`, `grep`, and `awk` to create the data file, `llenv.el`. After `awk` exits we inspect its exit status, `$?`, and only if it is 0, we define and export the new environmental variable, `ENV_SECOND_SUBMISSION`, and the script resubmits itself, because that is what `$LOADL_STEP_COMMAND` evaluates to. The variable `ENV_SECOND_SUBMISSION` will be visible in the second instantiation of the job, because of the LoadLeveler `#@environment=COPY_ALL` directive.

If the environmental variable `ENV_SECOND_SUBMISSION` is found to have been set to a non-zero string, the second clause of the if statement is executed. Within that clause we invoke emacs on the `llenv.el` file. The file is removed after emacs exits.

Observe that the `#@output` and `#@error` directives have been defined in terms of `$(job_name)` this time. Each instantiation of the script will have a different `$(job_name)`, so that the output and error files for the second instantiation of the job will not overwrite output and error files written by the first instantiation of the job. That is important in case any execution problems arise.

Using LoadLeveler Steps

If you want to execute a number of very simple tasks, in a sequence of LoadLeveler steps, tasks which do not involve much, if any, shell scripting, you may prefer to use LoadLeveler's *own* multiple job steps facility. That facility is a little bit tricky, and, in particular, you should not try to mix LoadLeveler steps with your own self-submitting shell scripts, because that may easily lead to confusion. In particular, remember, that if you do not use the LoadLeveler keyword `#@executable`, and thus, according to LoadLeveler's semantics, the LoadLeveler script itself becomes the executable, when the script is passed to, say, `ksh` for execution, all LoadLeveler keywords will be stripped, and the whole script will be executed in one go, even if the user has separated portions of the script with multiple `#@queue` directives.

Consider the following LoadLeveler job description file:

```
#
# Common definitions for all three steps
#
# @ output = $(job_name).$(step_name).out
# @ error = $(job_name).$(step_name).err
# @ job_type = serial
# @ class = test
# @ notification = always
# @ environment = COPY_ALL
# @ job_name = hello
#
# The first step: compile the program.
#
# @ step_name = compile
# @ executable = /afs/ovpit.indiana.edu/@sys/gnu/bin/gcc
# @ arguments = -o hello hello.c
# @ queue
```

```
#
# The second step: run the program if the compilation was successful.
#
# @ step_name = run
# @ dependency = compile == 0
# @ executable = /afs/ovpit.indiana.edu/@sys/gnu/bin/bash
# @ arguments = -c "exec hello"
# @ queue
#
# The third step: remove the binary if the run was successful.
#
# @ step_name = clean
# @ dependency = run == 0
# @ executable = /afs/ovpit.indiana.edu/@sys/gnu/bin/rm
# @ arguments = -e hello
# @ queue
```

When this script is submitted to LoadLeveler, three jobs will be placed in the queue. Initially two of those jobs will *wait* until the first job finishes execution. Then the second job will commence execution and the third will continue waiting. Finally, the third job will run. I should add that the second and the third jobs will run only if their direct ancestor has exited without any problems, leaving the exit status set to 0 behind.

The script is conceptually divided into four chunks.

The first chunk is a preamble with definitions common to all three job steps.

The second chunk describes the first step: it invokes the GNU C compiler and compiles a C program `hello.c` generating a binary `hello`, if the compilation has been successful.

The third chunk describes the second step: it will run only if the first step has left exit status 0 behind. That's what the directive

```
# @ dependency = compile == 0
```

is about. Observe a small complication. Instead of defining

```
# @ executable = hello
```

I have defined

```
# @ executable = /afs/ovpit.indiana.edu/@sys/gnu/bin/bash
# @ arguments = -c "exec hello"
```

The reason for this is that when the script is originally submitted to LoadLeveler, the file `hello` doesn't exist yet. So if I defined here `#@executable = hello` LoadLeveler would refuse the job and flag an error. All executables specified with the `#@executable` keyword must exist at the time the LoadLeveler script is submitted. The remedy is to specify my login shell as the executable instead, and then substitute (with `exec`) the shell with the binary produced in the first step.

The fourth chunk describes the third step: it will run only if the second step has left exit status 0 behind. That's what the directive

```
# @ dependency = run == 0
```

achieves. It is your responsibility, as a programmer, to ensure that this is indeed the case when your program exits cleanly.

This step removes the binary generated by the first step. The command `rm` is invoked with the `-e` option which will leave a trace on the `hello.clean.err` file:

```
rm: Removing hello
```

Can the same be achieved with shell scripting? Although I have warned you about possible pitfalls when mixing scripting and LoadLeveler steps, it is OK to do so, as long as your script does not attempt to resubmit itself. You might even consider the latter, but in that case you must carefully scrutinise the logic of both the shell script and the overlaying LoadLeveler script. Things may become easily convoluted, but not necessarily incorrect! Also, you should remember that the first occurrence of the keyword `#@executable` will override the shell script for all consecutive steps. If a shell script is present in the LoadLeveler command file, all steps defined before the first occurrence of the keyword `#@executable` will see the same script. Consequently, the script itself must be able to recognise which particular step is being executed during its instantiation and differentiate its actions accordingly. That information can be obtained from the environmental variable `LOADL_STEP_NAME`.

Here is an example of a 3-step LoadLeveler job, equivalent to the one discussed above, in which the actions are specified entirely using a shell script rather than three different `#@executables`.

```
# @ shell = /afs/ovpit.indiana.edu/@sys/gnu/bin/bash
# @ output = $(job_name).$(step_name).out
# @ error = $(job_name).$(step_name).err
# @ job_type = serial
# @ class = test
# @ notification = never
# @ environment = COPY_ALL
# @ job_name = hello
#
# @ step_name = compile
# @ queue
#
# @ step_name = run
# @ dependency = compile == 0
# @ queue
#
# @ step_name = clean
# @ dependency = run == 0
# @ queue
#
echo step: $LOADL_STEP_NAME
case $LOADL_STEP_NAME in
   compile )
      gcc -v -o hello hello.c 2>&1 ;;
   run )
      hello ;;
   clean )
      rm -e hello 2>&1 ;;
esac
```

7.4.6 Submitting Parallel Jobs

From the point of view of LoadLeveler there are three basic classes of parallel jobs: POE jobs, PVM3.3 jobs, and other parallel jobs. Of these LoadLeveler knows best how to run POE jobs - these can be MPI, MPL, HPF, PVMe, and Linda jobs. They are all based on a concept of a static processor pool, i.e., processors are assigned to the job at the beginning of its execution, and no additionall processors can be grabbed by the job while it runs. Consequently, certain PVM concepts such as dynamic allocation and de-allocation of parallel machines are not supported. This applies also to the next class of parallel jobs that LoadLeveler knows about: PVM3.3 jobs. LoadLeveler knows how to start PVM3.3 daemons on allocated machines, and how to invoke a PVM3.3 application.

LoadLeveler has no idea how to run other parallel jobs, e.g., network-Linda jobs (the parallel Gaussian falls in this category), ISIS jobs, LAM jobs, etc. But sometimes LoadLeveler can be fooled into thinking that these are either POE or PVM3.3 jobs, and, at the very least it will produce a list of allocated nodes, which user programs or daemons can then distribute themselves over.

POE jobs are easiest to run under LoadLeveler. There is always the same `#@executable=/usr/bin/poe` involved, regardless of whether the job is an MPI, MPL, HPF, or a Linda job. Only for PVMe jobs the executable is different: `#@executable=/usr/lpp/pvme/bin/pvmd3e`. The only difference between, say, an MPI and an HPF job is the compiler, that's been used to produce the POE binary.

At the very least POE jobs on the SP system should be specified by the following LoadLeveler keywords:

environment there is a number of POE related environmental variables, which specify, e.g., dynamic libraries to be used for the run

min_processors

max_processors

requirements switch interface specifications go here, e.g., `Adapter==hps_ip`, which means "use IP protocol over the switch".

job_type = parallel

Submitting POE/MPI Jobs

Here is an example of an MPI version of "hello world". The program itself looks as follows:

```
#include <stdio.h>
#include <mpi.h>

main(argc, argv)
int argc;
char *argv[];
{
        char name[BUFSIZ];
        int length;

        MPI_Init(&argc, &argv);
        MPI_Get_processor_name(name, &length);
        printf("%s: hello world\n", name);
        MPI_Finalize();
}
```

Compile it with the command

```
gustav@sp20:../LoadLeveler 17:07:04 !513 $ mpcc mpi-hello.c -o mpi-hello
gustav@sp20:../LoadLeveler 17:07:23 !514 $
```

and run by submitting the following LoadLeveler script:

```
gustav@sp20:../LoadLeveler 19:27:47 !647 $ cat mpi-hello.ll
# @ job_type = parallel
# @ environment = COPY_ALL; MP_EUILIB=ip; MP_INFOLEVEL=3
# @ requirements = (Adapter == "hps_ip")
# @ min_processors = 4
# @ max_processors = 8
# @ class = test
# @ notification = always
# @ executable = /usr/bin/poe
# @ arguments = mpi-hello
# @ output = mpi-hello.out
# @ error = mpi-hello.err
# @ queue
gustav@sp20:../LoadLeveler 19:27:49 !648 $ llsubmit mpi-hello.ll
```

This is what the file mpi-hello.out may look like after the run is finished:

```
gustav@sp20:../LoadLeveler 19:27:49 !648 $ cat mpi-hello.out
sp21.ucs.indiana.edu: hello world
sp19.ucs.indiana.edu: hello world
sp24.ucs.indiana.edu: hello world
sp20.ucs.indiana.edu: hello world
sp22.ucs.indiana.edu: hello world
sp23.ucs.indiana.edu: hello world
sp17.ucs.indiana.edu: hello world
sp18.ucs.indiana.edu: hello world
gustav@sp20:../LoadLeveler 19:29:11 !649 $
```

The file mpi-hello.err contains messages from poe:

```
gustav@sp20:../LoadLeveler 19:29:11 !649 $ cat mpi-hello.err
INFO: DEBUG_LEVEL changed from 0 to 1
D1<L1>: ./host.list file did not exist
```

```
D1<L1>: mp_euilib = ip
D1<L1>: node allocation strategy = 1
ATTENTION: 0031-408  8 nodes allocated by LoadLeveler, continuing...
INFO: 0031-119  Host sp21.ucs.indiana.edu allocated for task 0
INFO: 0031-119  Host sp24.ucs.indiana.edu allocated for task 1
INFO: 0031-119  Host sp19.ucs.indiana.edu allocated for task 2
INFO: 0031-119  Host sp22.ucs.indiana.edu allocated for task 3
INFO: 0031-119  Host sp20.ucs.indiana.edu allocated for task 4
INFO: 0031-119  Host sp23.ucs.indiana.edu allocated for task 5
INFO: 0031-119  Host sp17.ucs.indiana.edu allocated for task 6
INFO: 0031-119  Host sp18.ucs.indiana.edu allocated for task 7
...
```

Observe that since there was no file `host.list` in my working directory, LoadLeveler consulted the Resource Manager daemons and allocated the nodes for the run on its own. Only nodes from the group supporting the `test` class, in this case, would be selected.

If you need to perform certain manipulations before and after running the main executable you can use scripting the same way as has already been discussed for sequential programs. For example, you could run the MPI `"hello world"` example as follows:

```
gustav@sp20:../LoadLeveler 19:35:59 !668 $ cat mpi-hello-2.ll
# @ shell = /afs/ovpit.indiana.edu/@sys/gnu/bin/bash
# @ job_type = parallel
# @ environment = COPY_ALL; MP_EUILIB=ip; MP_INFOLEVEL=3
# @ requirements = (Adapter == "hps_ip")
# @ min_processors = 4
# @ max_processors = 8
# @ output = mpi-hello.out
# @ error = mpi-hello.err
# @ class = test
# @ notification = never
# @ queue
poe mpi-hello
gustav@sp20:../LoadLeveler 19:36:10 !669 $
```

Observe that I have instructed LoadLeveler *not* to send me any e-mail at all, regardless of what is going to happen to the job. You may wish to use `#@notification=never` if you submit a lot of short jobs to the `test` class.

The next example shows how you can have even more control over the way your POE job is run under LoadLeveler:

```
gustav@sp20:../LoadLeveler 19:41:24 !683 $ cat mpi-hello-3.ll
# @ shell = /afs/ovpit.indiana.edu/@sys/gnu/bin/bash
# @ job_type = parallel
# @ environment = COPY_ALL;
# @ requirements = (Adapter == "hps_ip")
# @ min_processors = 4
# @ max_processors = 8
# @ output = mpi-hello.out
# @ error = mpi-hello.err
# @ class = test
# @ notification = never
# @ queue
```

```
> host.list.$LOADL_STEP_ID
NPROC=0
for node in $LOADL_PROCESSOR_LIST
do
   echo $node >> host.list.$LOADL_STEP_ID
   NPROC=`expr $NPROC + 1`
done
#
export MP_HOSTFILE=host.list.$LOADL_STEP_ID
export MP_PROCS=$NPROC
export MP_EUILIB=ip
export MP_EUIDEVICE=css0
export MP_INFOLEVEL=3
#
poe mpi-hello
#
rm $MP_HOSTFILE
gustav@sp20:../LoadLeveler 19:41:28 !684 $
```

This time I construct the POE host file dynamically and pass it to POE via the `MP_HOSTFILE` environmental variable. At the same time I count the number of allocated nodes. I have requested that number to be between 4 and 8, but the exact number will be known only when the job starts. That number is passed to POE via the `MP_PROCS` environmental variable.

Here is what the `mpi-hello.err` file produced by the run looks like:

```
gustav@sp20:../LoadLeveler 19:43:58 !685 $ cat mpi-hello.err
INFO: DEBUG_LEVEL changed from 0 to 1
D1<L1>: Open of file host.list.sp20.ucs.indiana.edu.58.0 successful
D1<L1>: mp_euilib = ip
D1<L1>: node allocation strategy = 1
INFO: 0031-119  Host sp18.ucs.indiana.edu allocated for task 0
INFO: 0031-119  Host sp22.ucs.indiana.edu allocated for task 1
INFO: 0031-119  Host sp19.ucs.indiana.edu allocated for task 2
INFO: 0031-119  Host sp21.ucs.indiana.edu allocated for task 3
INFO: 0031-119  Host sp17.ucs.indiana.edu allocated for task 4
INFO: 0031-119  Host sp23.ucs.indiana.edu allocated for task 5
INFO: 0031-119  Host sp20.ucs.indiana.edu allocated for task 6
INFO: 0031-119  Host sp24.ucs.indiana.edu allocated for task 7
...
```

The listing shows that host names were obtained from the dynamically generated host file, and it also shows which task runs on which host.

Submitting POE/HPF Jobs

High Performance Fortran jobs are POE jobs, and as such they are run under LoadLeveler or interactively in the same way as POE/MPI jobs (see section 7.4.6).

7.5 Checkpointing and Resubmission

In this section we will discuss how to time, checkpoint and automatically resubmit LoadLeveler jobs. Rather than relying on a special LoadLeveler mechanism for checkpointing jobs, which requires linking your program with LoadLeveler libraries, and which does not work for parallel jobs, and for other batch queueing systems, such as NQS, here I demonstrate how you can easily implement your own timing, checkpointing, and job resubmission mechanism in C, Fortran 90, and in Common Lisp.

The procedures discussed in this section are not limited to LoadLeveler. They should work for any batch submission system, as long as the batch jobs are described in terms of shell scripts, and as long as the system in question is IEEE-1003 (POSIX) compliant. They are applicable both to sequential and parallel jobs.

There are four issues that need to be addressed when automatically checkpointing and resubmitting your LoadLeveler jobs.

- **Timing the job:** Your job must know how much CPU or wall-clock time it used so far, and how much time there is still left.
- **Saving the state of the job:** This usually involves dumping a data file which contains an essential summary of the state of the system that is being computed. That file will be read when the job is restarted, and computation will commence from the point reached when the file has been dumped.
- **Informing the parent process (usually a shell) that the computation should be continued:** This can be done, for example, by exiting the job with a non-zero exit status. Alternatively you could write a specific message on a log file (to be searched for by the shell script when the job exits) or create an empty flag file.
- **Resubmitting the job:** Depending on whether the job should be continued, the LoadLeveler script, before exiting, should either
  1. resubmit itself, possibly with certain new flags or variables set up, or
  2. clean up and inform the user that the computation has been completed.

7.5.1 Timing a Job

Timing a Job in C

Probably most C-language programmers know how to time their jobs, because functions `time` and `clock` are parts of the standard C library, which is defined by ANSI C specifications.

Function `time` takes a pointer to `time_t` as an argument and returns a value of `time_t` on exit. On our system `time_t` is defined on `/usr/include/sys/types.h` and /usr/include/time.h as long. If the pointer is not NULL, the return value is also placed in whatever location the pointer points at. The returned value is the current calendar time, in seconds, since the Epoch, i.e., 00:00:00 GMT, 1st of January 1970: popularly celebrated as the day when UNIX was born.

You would use function time in order to find out about the elapsed wall-clock time. If you know that, say, your queue allows only up to two wall-clock hours (7200 seconds) per job, by checking how much time you've used so far, you will know how much time there is still left too.

Function clock does not take any arguments and returns a value ot type clock_t, which is defined on /usr/include/sys/types.h and /usr/include/time.h as int. This function returns CPU time that elapsed since the execution of the program commenced. The returned time is not in seconds. It is in clock cycles. There is a constant CLOCKS_PER_SEC defined on /usr/include/time.h, which tells how many clock cycles there are per second. So, in order to find out how many CPU seconds you have used so far you have to divide the result obtained by calling clock by CLOCKS_PER_SEC.

Because function clock returns clock_t, i.e., int (on AIX), it should be called frequently. Once the returned value reaches MAX_INT, which is defined on /usr/include/values.h, the clock resets itself and resumes counting from 0.

The following example illustrates how to use functions time and clock.

```
#include <sys/types.h>
#include <time.h>
#include <unistd.h>
#include <stdio.h>
#include <math.h>

main()
{
  time_t  t0, t1; /* time_t is defined on <time.h> and <sys/types.h> as long */
  clock_t c0, c1; /* clock_t is defined on <time.h> and <sys/types.h> as int */

  long count;
  double a, b, c;

  printf ("using UNIX function time to measure wallclock time ... \n");
  printf ("using UNIX function clock to measure CPU time ... \n");

  t0 = time(NULL);
  c0 = clock();

  printf ("\tbegin (wall):            %ld\n", (long) t0);
  printf ("\tbegin (CPU):             %d\n", (int) c0);

  printf ("\t\tsleep for 5 seconds ... \n");
  sleep(5);

  printf ("\t\tperform some computation ... \n");
  for (count = 1l; count < 10000000l; count++) {
     a = sqrt(count);
     b = 1.0/a;
     c = b - a;
  }
  t1 = time(NULL);
  c1 = clock();

  printf ("\tend (wall):              %ld\n", (long) t1);
  printf ("\tend (CPU);               %d\n", (int) c1);
  printf ("\telapsed wall clock time: %ld\n", (long) (t1 - t0));
  printf ("\telapsed CPU time:        %f\n", (float) (c1 - c0)/CLOCKS_PER_SEC);
}
```

Compile this program with

```
gustav@sp20:../time 21:55:32 !861 $ gcc -o c-time c-time.c -lm
gustav@sp20:../time 21:55:40 !862 $
```

and run it as follows

```
gustav@sp20:../time 21:55:40 !862 $ time -p ./c-time
using UNIX function time to measure wallclock time ...
using UNIX function clock to measure CPU time ...
        begin (wall):            916801059
        begin (CPU):             0
                sleep for 5 seconds ...
                perform some computation ...
        end (wall):              916801070
        end (CPU);               5970000
        elapsed wall clock time: 11
        elapsed CPU time:        5.970000
Command exited with non-zero status 35
real 10.98
user 5.97
sys 0.01
gustav@sp20:../time 21:57:50 !863 $
```

Observe that times returned by this program agree with times returned by the GNU command `time`, which on our system lives in

```
/afs/ovpit.indiana.edu/@sys/gnu/bin
```

Timing a Job in Fortran

Even Fortran-90 does not provide portable means for checking CPU time usage without calling vendor and operating system specific routines.

Fortran 90 defines two intrinsic procedures `date_and_time` and `system_clock`, which return elapsed wall-clock time in various formats.

The `date_and_time` procedure takes 4 arguments, all of which are optional:

date: intent(out) a character string at least 8 characters long

time: intent(out) a character string at least 10 characters long

zone: intent(out) a character string at least 5 characters long

values: intent(out) an array of integers at least 8 entries long

For our purposes we don't need date or time returned as strings. We only need the numbers, which are returned in values, so we'll call this procedure using a keyword argument list:

```
call date_and_time (values=time_array)
```

where `time_array` is our array of integers. The returned values will have the following ordering:

time_array(1) year

time_array(2) month of the year

time_array(3) day of the month

time_array(4) time offset with respect to UTC in minutes

time_array(5) hour of the day

time_array(6) minutes of the hour

time_array(7) seconds of the minute

time_array(8) milliseconds of the second

Subroutine `system_clock` is somewhat easier to use. It takes 3 optional arguments:

count, intent(out) an integer

count_rate: intent(out) an integer

count_max: intent(out) an integer

This function is somewhat similar to C-function `clock`, in the sense that it counts time at a rate of `count_rate` counts per second up to `count_max`, and then resets itself to zero and resumes the counting. But unlike `clock` this function measures wall-clock time, not the CPU time. As you will see from the following example, under AIX procedure `system_clock` resets every day at midnight. But this particular behaviour is not specified in F90 standard.

In fact there is no intrinsic Fortran-90 procedure for measuring CPU time. For that we have to use XL-Fortran service and utility function `etime_`. At this stage the program ceases to be portable, so it is a good idea to isolate the parts of the code that rely on `etime_` with cpp `#ifdef .. #endif` brackets. In the example below I use `gcc -E -P -C` instead of cpp. It is important to remove cpp generated line references before passing the file to Fortran compiler. Option `-P` ensures that. The importance of option `-C` will become clearer in our next Fortran-90 example.

Function `etime_` is defined in the `xlfutility` module, which must be included with the `use` statement:

```
use xlfutility
```

The function takes a structure of type `tb_type` as argument `(intent(out))` and returns the sum of system and user components of the CPU time since the start of the execution of a process. Additionally user time and system time are written on `usrtime` and `systime` slots of the argument.

For more information about service and utility procedures provided in `xlfutility` read the "XL Fortran for AIX, Language Reference, Version 3, Release 2" manual, pages 445-451. The manual, in compressed PostScript, can be found in the /usr/lpp/xlf/ps directory on any SP node.

The following Fortran 90 program shows how to use all three procedures in order to time your computation.

```
      program f_time

#ifdef XLF
      use xlfutility

! Variables for function dtime_

      real(4) elapsed_0, elapsed_1
      type (tb_type) etime_struct_0, etime_struct_1
#endif

! Variables for subroutine system_clock

      integer count_0, count_1, count_rate, count_max

! Variables for subroutine date_and_time

      integer time_array_0(8), time_array_1(8)
      real start_time, finish_time

! Variables for computation

      integer n
      parameter (n = 1000000)
      double precision a(n), b(n), c(n)

      write (6, '(1x, 1a)') 'using F90 procedure date_and_time ...'
      write (6, '(1x, 1a)') 'using F90 procedure system_clock ...'
#ifdef XLF
      write (6, '(1x, 1a)') 'using XLF function dtime_ ...'
#endif

! Mark the beginning of the program

      call date_and_time(values=time_array_0)
      start_time = time_array_0 (5) * 3600 + time_array_0 (6) * 60 &
           + time_array_0 (7) + 0.001 * time_array_0 (8)
      call system_clock(count_0, count_rate, count_max)
#ifdef XLF
      elapsed_0 = etime_(etime_struct_0)
#endif
```

```
      write (6, '(8x, 1a, 1f16.6)') 'begin (date_and_time):  ', &
           start_time
      write (6, '(8x, 1a, 1f16.6)') 'begin (system_clock):   ', &
           count_0 * 1.0 / count_rate
#ifdef XLF
      write (6, '(8x, 1a, 1f16.6)') 'begin (etime_%usrtime): ', &
           etime_struct_0%usrtime
      write (6, '(8x, 1a, 1f16.6)') 'begin (etime_%systime): ', &
           etime_struct_0%systime
#endif

! Sleep for 5 seconds

#ifdef XLF
      write (6, '(16x, 1a)') 'sleep for 5 seconds ... '
      call sleep_ (5)
#endif

! Perform some computation

      write (6, '(16x, 1a)') 'perform some computation ... '
      a = (/ (i, i = 1, n) /)
      a = sqrt(a)
      b = 1.0 / a
      c = b - a

! Mark the end of the program

      call date_and_time(values=time_array_1)
      finish_time = time_array_1 (5) * 3600 + time_array_1 (6) * 60 &
           + time_array_1 (7) + 0.001 * time_array_1 (8)
      call system_clock(count_1, count_rate, count_max)
#ifdef XLF
      elapsed_1 = etime_(etime_struct_1)
#endif

      write (6, '(8x, 1a, 1f16.6)') 'end (date_and_time):    ', &
           finish_time
      write (6, '(8x, 1a, 1f16.6)') 'end (system_clock):     ', &
           count_1 * 1.0 / count_rate
#ifdef XLF
      write (6, '(8x, 1a, 1f16.6)') 'end (etime_%usrtime):   ', &
           etime_struct_1%usrtime
      write (6, '(8x, 1a, 1f16.6)') 'end (etime_%systime):   ', &
           etime_struct_1%systime
#endif

! Print elapsed time

      write (6, '(8x, 1a, 1f16.6)') 'elapsed wall clock time:', &
           finish_time - start_time
#ifdef XLF
      write (6, '(8x, 1a, 1f16.6)') 'elapsed CPU time:       ', &
           etime_struct_1%usrtime - etime_struct_0%usrtime
#endif

      end program f_time
```

This file must be passed through cpp first in order to generate the plain Fortran code. Then the code must be compiled and linked with Fortran-90 compiler. The most convenient way to go about all that is to write appropriate instructions on a Makefile and use make to generate the binary. Here is the Makefile that you can use for this F90 example code:

```
F90     = xlf90
CPP     = gcc -E -P -C
DEFINES = -DXLF
OPTS    = # -g

all: f_time

f_time: f_time.o
        $(F90) $(OPTS) -o f_time f_time.o

f_time.o: f_time.f
        $(F90) $(OPTS) -c f_time.f

f_time.f: f_time.cpp
        $(CPP) $(DEFINES) f_time.cpp > f_time.f

clean:
        rm -f f_time.f f_time.o f_time
```

On the IU SP system I have compiled and run this program as follows:

```
<6:05:00 !705 $ gcc -E -P -C -DXLF f_time.cpp > f_time.f
gustav@sp19:../LoadLeveler 16:05:15 !706 $ xlf90 -o f_time f_time.f
** f_time   === End of Compilation 1 ===
1501-510  Compilation successful for file f_time.f.
gustav@sp19:../LoadLeveler 16:05:26 !707 $ time -p ./f_time
 using F90 procedure date_and_time ...
 using F90 procedure system_clock ...
 using XLF function dtime_ ...
        begin (date_and_time):      57937.957031
        begin (system_clock):       57937.949219
        begin (etime_%usrtime):         0.000000
        begin (etime_%systime):         0.010000
                sleep for 5 seconds ...
                perform some computation ...
        end (date_and_time):        57944.312500
        end (system_clock):         57944.308594
        end (etime_%usrtime):           0.850000
        end (etime_%systime):           0.210000
        elapsed wall clock time:        6.355469
        elapsed CPU time:               0.850000
real 6.43
user 0.85
sys 0.21
gustav@sp19:../LoadLeveler 16:05:44 !708 $
```

And, as before, we can see that our internal estimates agree pretty well with results returned by UNIX program time. There is a small discrepancy of 0.08 s in the estimate of the elapsed wall-clock time (6.35s versus 6.43s), the explanation of which is left to the reader as an exercise. Also observe that time returned by procedure `system_clock` is roughly the same as time returned by procedure `date_and_time`, which means that `system_clock` must be reset at midnight, as I have already remarked above.

7.5.2 Restoring and Saving the State of a Job

In this section we shall discuss how to save and then restore the state of the computation between successive invocations of a program via LoadLeveler. The basic idea is that the only way any information can be transferred between successive invocations of a program is either

1. through a file, or
2. through an environmental variable, or
3. through command line switches

Transferring data through a file is perhaps the most common practice. Using files you can transfer very large amounts of data: e.g., the whole state of a 3D flow, or the whole state of a protein, or the whole state of a car in a crash simulation. Basically, files can be used to transfer any information from one instantiation of a program to another, including even small items of information, such as whether the program should restart a computation from a previously reached state, or whether it should start a new computation.

Instead of writing on files, the program, in principle, can also write on user's environment. On next invocation the program can check for existence and state of certain predefined environmental variables, and obtain required information that way. This method is good for transferring small amounts of information, e.g., the name of a checkpoint file, or the request to initialise a run, but not for very large data sets.

Of course, using environmental variables will not work if the variables themselves are not transmitted from one LoadLeveler process to another one. There is a special LoadLeveler directive:

```
# @ environment = COPY_ALL
```

which instructs LoadLeveler to copy all environmental variables from the current shell and transfer them to the shell within which the job will be executed.

But there is one problem with writing on user's environment. This can be done portably only from within C (or C++). Fortran-90 provides an intrinsic procedure for reading environment, getenv, but not for writing on it. Common Lisp, in turn, specifies only that such procedures should be available in the implementation dependent system (nickname: sys) package, but does not specify exactly what should be in that package. Most Lisps, that I know of, have `sys::getenv`, but, again, not all of them have `sys::putenv` or `sys::setenv`.

So, we shall have to use some other mechanism to convey information about the name of the checkpoint file and whether the job should be continued, for example, we can write it at the end of a log file. After our application exits, the LoadLeveler script responsible for the execution of the application can inspect the log, and if it finds the instruction that the job should be continued, it can transfer that information to environmental variables, and resubmit itself. When LoadLeveler again gets to activate that script, our application will begin by checking for certain variables in the environment and for their content. From there it will learn if it should continue or reinitialise the computation, and if it should continue, where it should look for information about the state reached by the previous run.

Instead of using environment we could transfer the instruction to restart the computation and the name of the checkpoint file by using command line arguments. This is a neat way of doing things, but it's somewhat harder to program than reading the environment. You can use this mechanism portably with C and C++, but not with Fortran or Lisp programs. Almost all Fortrans, that I have worked with, support reading command line arguments, but they all do it differently, and, as I said, it's not a part of Fortran standard.

Restoring and Saving in C

The following listing shows a very simple C-language program which, if requested, reads the state of computation from a file. If not requested it initialises a new computation. Then some further computation is performed and the new state is again saved on a file.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

main()
{
  char *restart_name, *restart, old_restart_name[BUFSIZ];
  FILE *restart_file;
  int n;

  /* Is this a continued job or a new one? */

  if (! (restart = getenv ("RSAVE_RESTART"))) {
    printf ("Starting a new run.\n");
    n = 0;
  }
  else {
    if (! (restart_name = getenv ("RSAVE_CHECKFILE"))) {
      fprintf (stderr, "error: no checkpoint file for the restart job\n");
      exit (1);
    }
    else {
      printf ("Restarting the job from %s.\n", restart_name);
      if (! (restart_file = fopen(restart_name, "r"))) {
        perror (restart_name);
        exit (2);
      }
      else {
        if (! (fscanf (restart_file, "%d", &n) > 0)) {
          fprintf (stderr, "%s: input file format error\n", restart_name);
          exit (3);
        }
        else {
          fclose (restart_file);
        }
      }
    }
  }

  printf ("n = %d\n", n);
  printf ("\tcomputing ... "); fflush (stdout);
  sleep (5);
  n++;
  printf ("done.\n");
  printf ("n = %d\n", n);

  if (! (restart_name = getenv ("RSAVE_CHECKFILE"))) {
    printf ("checkpointing not requested, exiting...\n");
    exit (0);
  }
  else {
    if (restart) {
      strcpy (old_restart_name, restart_name);
      strcat (old_restart_name, ".old");
      printf ("renaming old restart file to %s\n", old_restart_name);
      if (0 > rename (restart_name, old_restart_name)) {
        perror (old_restart_name);
        exit (4);
      }
    }
    printf ("saving data on %s\n", restart_name);
    if (! (restart_file = fopen (restart_name, "w"))) {
      perror (restart_name);
      exit (5);
    }
    else {
      fprintf (restart_file, "%d\n", n);
      fclose (restart_file);
    }
  }
  exit (0);
}
```

I'll explain how this program works in detail below, but first let's just see what it does:

```
gustav@sp19:../LoadLeveler 13:39:41 !516 $ env | grep RSAVE
RSAVE_CHECKFILE=rsave.dat
RSAVE_RESTART=yes
gustav@sp19:../LoadLeveler 13:39:45 !517 $ unset RSAVE_RESTART
gustav@sp19:../LoadLeveler 13:39:51 !518 $ ./rsave
Starting a new run.
n = 0
```

```
        computing ... done.
n = 1
saving data on rsave.dat
gustav@sp19:../LoadLeveler 13:40:03 !519 $ export RSAVE_RESTART="yes"
gustav@sp19:../LoadLeveler 13:40:14 !520 $ ./rsave
Restarting the job from rsave.dat.
n = 1
        computing ... done.
n = 2
renaming old restart file to rsave.dat.old
saving data on rsave.dat
gustav@sp19:../LoadLeveler 13:40:22 !521 $ cat rsave.dat
2
gustav@sp19:../LoadLeveler 13:40:29 !522 $ ./rsave
Restarting the job from rsave.dat.
n = 2
        computing ... done.
n = 3
renaming old restart file to rsave.dat.old
saving data on rsave.dat
gustav@sp19:../LoadLeveler 13:40:51 !523 $ ./rsave
Restarting the job from rsave.dat.
n = 3
        computing ... done.
n = 4
renaming old restart file to rsave.dat.old
saving data on rsave.dat
gustav@sp19:../LoadLeveler 13:41:28 !524 $
```

Here is the promised explanation of the program in detail.

The first thing that the program does, is to check for the existence of the environmental variable `RSAVE_RESTART`. If the variable does not exist, the program starts a new run and initialises `n` to 0.

If the variable `RSAVE_RESTART` exists (it doesn't really matter what is its value) then we first check if another variable, which should specify the name of the checkpoint file, `RSAVE_CHECKFILE`, exists too. If it doesn't, then we have no way to find the name of the checkpoint file. So in that case we print an error message, flag an error on exit (value 1) and exit.

If the variable `RSAVE_CHECKFILE` exists then we use its value as the name of the checkpoint file, print a message about restarting the job from that file, and attempt to open it for reading.

If for some reason the file cannot be opened, we print the diagnostic on standard output (with `perror`), flag an error (value 2) and exit.

If the file has been opened without problems we try to read an integer number from it. That integer is the whole object of our simple computation in this program and it represents the state of the system.

It may happen that for some reason the checkpoint file does not contain that integer. In that case we print the corresponding error message, flag an error (value 3) and exit.

But if everything goes well, by this time we should have our state of the system in hand, so we close the checkpoint file (in case of an error exit the file would be closed automatically) and commence the computation.

The computation is quite trivial. We simply increment the integer read from the file by 1. In order to add a little more body to the program we also sleep for 5 seconds (this is called putting on weight). We will need that sleep in our next example, which will combine timing with saving and restoring.

Once the computation is finished we again check the environmental variable `RSAVE_CHECKFILE`. Observe that this variable has not been looked up so far by the branch of the program, that does the initialisation. That is why we do it here again, even though the other branch, which is responsible for the restarting of the job, would have looked it up already.

If the variable `RSAVE_CHECKFILE` is not defined, we write the message that "checkpointing has not been requested" and exit. No error condition is flagged this time.

If the variable `RSAVE_CHECKFILE` exists, and if the job is a restarted one, then we attempt to rename the original restart file to whatever its old name was with a suffix `.old` appended.

If for some reason that cannot be done, we print diagnostic on standard error using `perror`, flag an error (value 4) and exit.

Otherwise, having renamed the old restart file, we attempt to open, this time for writing, a new file bearing the old name. If for some reason that cannot be done a diagnostic is printed on standard error with `perror`, an error exit is flagged (value 5) and the program aborts.

Otherwise, i.e., if all went well and we have the new restart file opened, we write the new value of `n` on it, close it, and exit with status 0.

This is really quite simple stuff. Whatever complexity there is in the presented example, it derives from my attempt to make the program robust. Regardless of whether variables `RSAVE_RESTART` and `RSAVE_CHECKFILE` exist, regardless of whether the data file itself exists, the program should always do something more or less sensible, write meaningful error messages if need be, and exit gracefully conveying a meaningful exit value to the shell. For seasoned C and C++ programmers all that is just bread and butter.

Restoring and Saving in Fortran

Our Fortran-90 example does much the same as our C example, so if you have skipped the previous section, you should go back to it and read it now. Again, I have attempted to make the program relatively robust, which adds to its complexity a little.

In Fortran-90 procedures, which operate on files, are implemented as subroutines, not as functions. For this reason, the Fortran version of our example does not flow as smoothly as our C program. Any I/O problems must be addressed by jumping to a specified label. It is customary to place all error handlers together at the end of the file.

When the `cpp` preprocessor is invoked on this file, we must use the `-C` option, i.e., we must preserve C (and C++) language comments. The reason for that is that Fortran string-append operator, `//`, is the same as the C++ comment marker. Without the `-C` option, all string-append operations would be stripped from the produced Fortran code.

There is no way to rename a file within Fortran-90. So, in order to save the checkpoint file under a new name I have to use the intrinsic subroutine system. Unfortunately, the way this subroutine is implemented, no exit status is returned to the calling Fortran program, so we have no means of checking, if the requested operation was successful.

For this reason, when the checkpoint file is opened for writing, I use the 'replace' status. If the renaming operation is unsuccessful, the old checkpoint file will be replaced with the new one.

Although the package `xlfutility` provides subroutine `exit_`, there is no need to call it here. If the stop statement is followed by a number, XL Fortran makes that number available to the parent shell as the exit status of the program.

Observe that Fortran-90 makes life of a Fortran programmer a lot easier. One of the most useful new Fortran-90 facilities is function `len_trim`, which returns the real length of a string with trailing blanks stripped. In the `open` statement you'll find a new directive, `'action'`, which specifies the kind of operation that will be attempted on the file, e.g., `'read'` or `'write'`.

Now, here is the Fortran-90 example itself:

```
program rsave

#ifdef XLF
  use xlfutility
#endif
  character (len=64) restart, restart_name, old_restart_name
  character (len=512) command
  integer n, restart_file, status
  parameter (restart_file = 21)

! Is this a continued job or a new one?

  call getenv ('RSAVE_RESTART', restart)
  if (len_trim(restart) .eq. 0) then
     write (6, '(1x, 1a)') 'Starting a new run'
     n = 0
  else
     call getenv ('RSAVE_CHECKFILE', restart_name)
     if (len_trim(restart_name) .eq. 0) then
        write (6, '(1x, 1a)') 'Error: no checkpoint file for the restart job'
        stop 1
     else
        write (6, '(1x, 2a)') 'Restarting the job from ', restart_name
        open (unit=restart_file, iostat=status, err=100, file=restart_name, &
             status='old', action='read')
        read (restart_file, '(1i7)', iostat=status, err=110, end=110) n
        close (restart_file)
     end if
  end if

! This is our computation part

  write (6, '(1x, 1a, 1i7)') 'n = ', n
  write (6, '(9x, 1a, $)') 'computing ... '
#ifdef XLF
  call flush_ (6)
#endif

#ifdef XLF
  call sleep_ (5)
#endif
  n = n + 1
  write (6, '(1a)') 'done.'
  write (6, '(1x, 1a, 1i7)') 'n = ', n

! And now we save the result on a new checkpoint file, saving
! the old one under a new name if need be.

  call getenv ('RSAVE_CHECKFILE', restart_name)
  if (len_trim(restart_name) .eq. 0) then
     write (6, '(1x, 1a)') 'Checkpointing not requested, exiting ... '
     stop 0
  else
     if (.not. (len_trim(restart) .eq. 0)) then
        old_restart_name = restart_name (1:len_trim(restart_name)) // '.old'
        write (6, '(1x, 2a)') 'Renaming the old restart file to ', &
             old_restart_name
        command = 'mv' // ' ' // restart_name // ' ' // old_restart_name
        call system (command)
     end if
     write (6, '(1x, 2a)') 'Saving data on ', restart_name
     open (unit=restart_file, iostat=status, err=120, file=restart_name, &
             status='replace', action='write')
     write (restart_file, '(1i7)') n
     close (restart_file)
  end if
  stop 0

! error handlers

! error while opening the checkpoint file for reading

100 write (6, '(1x, 3a)') 'Error: while opening ', restart_name, ' for reading'
  write (6, '(8x, 1a, 1i7)') 'iostat = ', status
  stop 2

! error while trying to read input file

110 write (6, '(1x, 2a)') 'Error: while reading from ', restart_name
  write (6, '(8x, 1a, 1i7)') 'iostat = ', status
  stop 3

! error while opening the checkpoint file for writing

120 write (6, '(1x, 3a)') 'Error: while opening ', restart_name, ' for writing'
  write (6, '(8x, 1a, 1i7)') 'iostat = ', status
  stop 5

end program rsave
```

Compile this program as follows:

```
<57:39 !532 $ gcc -E -P -C -DXLF rsave.cpp > rsave.f
gustav@sp19:../LoadLeveler 13:57:46 !533 $ xlf90 -o rsave rsave.f
** rsave   === End of Compilation 1 ===
1501-510  Compilation successful for file rsave.f.
gustav@sp19:../LoadLeveler 13:58:11 !534 $
```

And run it like that:

```
gustav@sp19:../LoadLeveler 13:58:11 !534 $ env | grep RSAVE
RSAVE_CHECKFILE=rsave.dat
RSAVE_RESTART=yes
gustav@sp19:../LoadLeveler 13:59:10 !535 $ unset RSAVE_RESTART
gustav@sp19:../LoadLeveler 13:59:17 !536 $ ./rsave
 Starting a new run
 n =       0
        computing ... done.
 n =       1
 Saving data on rsave.dat
STOP 0
gustav@sp19:../LoadLeveler 13:59:25 !537 $ export RSAVE_RESTART=yes
gustav@sp19:../LoadLeveler 13:59:34 !538 $ ./rsave
 Restarting the job from rsave.dat
 n =       1
        computing ... done.
 n =       2
 Renaming the old restart file to rsave.dat.old
 Saving data on rsave.dat
STOP 0
gustav@sp19:../LoadLeveler 13:59:41 !539 $ ./rsave
 Restarting the job from rsave.dat
 n =       2
        computing ... done.
 n =       3
 Renaming the old restart file to rsave.dat.old
 Saving data on rsave.dat
STOP 0
gustav@sp19:../LoadLeveler 13:59:57 !540 $
```

7.5.3 Restoring, Timing, and Saving a Job: the Complete Application

In this section we shall combine job timing with job restoring and saving, and produce a complete application, which, in the next section, will be combined with a LoadLeveler script, so as to produce an automatically resubmitting job. As in the previous two sections we shall present example codes in C, and in Fortran-90.

The program is a slight modification of our restore and save example. There are no really new elements here, which would require a broader explanation.

The additional logic that is laid out on top of the restore and save example is as follows.

We begin by checking for a new environmental variable, `RSAVE_TIME_LIMIT`. If that variable does not exist then we assume that time allowed for this job is unlimited and things work more or less as before. If the variable exists, then we attempt to read its value assuming that it is going to be a number. If it is not a number we print an error message and exit. If it is a number, then the number is assigned to variable `time_limit` and assumed to represent the number of wall-clock seconds allocated to this job.

On our IU system those queues, which are timed at all, are timed in terms of CPU seconds. But what really matters to other users and to yourself is how long you have to wait until your job gets out of the way. For this reason I use wall-clock timers, i.e., function `time` in C and subroutine `system_clock` in Fortran 90. However, you can modify the programs easily to look up the CPU time instead.

Once the information about time limit is obtained we proceed exactly as before, until we get to the part of the program which does the computation. Instead of just incrementing number `n` and sleeping for 5 seconds, we enter a loop.

If no timing has been requested the loop keeps incrementing `n` and sleeping, until `n` becomes greater than `LAST_N`. The latter is an arbitrary constant, which in our toy example represents something like a convergence criterion. Once the convergence has been reached, the finished flag is set to `TRUE` and the loop exits.

Things are more interesting if timing of the job has been requested (by setting the environmental variable `RSAVE_TIME_LIMIT` to some number of seconds). In that case we measure time taken by one iteration of the loop, and we check how much time there is still left after the iteration has finished. If there is still enough time to perform another iteration we continue, if not, the loop exits.

Because saving the data, cleaning up, and executing LoadLeveler script may take additional time we have to include a `SAFETY_MARGIN` while calculating time that still remains. In this case we set `SAFETY_MARGIN` to 10 seconds, but if you have to save a very large data set, you should probably reserve a couple of minutes.

Flagging the resubmission is accomplished as follows.

Before exiting, we check if the whole job is finished, which it will be once the convergence criterion is satisfied. If the job is finished we write `FINISHED` on standard output. Otherwise we write `CONTINUE`.

If the standard output has been logged on a file, after the program exits, the LoadLeveler script can inspect the log, and resubmit itself, if it finds the word CONTINUE in the log. How that works will be shown in the next section.

The Complete Application in C

Here is the C version of the program. The wall-clock time is measured using the UNIX function `time`. The parameters `LAST_N` and `SAFETY_MARGIN` have been implemented as `cpp` constants. I could also read them from the environment, a command line, or from an input file, but that would clutter the example.

The program always executes the statements of the `do ... while` loop at least once, because the exit condition is tested at the end of the loop.

Observe that the variable `quit_time` is initialised to `1l`. That way, if timing is not requested, it remains always positive and the `while` test fires up only when the job is finished. Furthermore the variable `timing` is initialised to `TRUE`, and becomes `FALSE` only if there is no environmental variable `RSAVE_TIME_LIMIT`. The job is assumed to be unfinished on entry (the variable finished is initialised to `FALSE`) and becomes finished only when `n` becomes greater than `LAST_N`. This means that once `n` becomes greater than `LAST_N`, you can still submit the job and it will always increment `n` by 1 before exiting.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#ifndef TRUE
# define TRUE 1
#endif
#ifndef FALSE
# define FALSE 0
#endif

#ifndef LAST_N
# define LAST_N 30
#endif

#ifndef SAFETY_MARGIN
# define SAFETY_MARGIN 10
#endif

main()
{
  char *restart_name, *restart, old_restart_name[BUFSIZ];
  FILE *restart_file;
  int n, finished = FALSE, timing = TRUE;
  time_t t0, t1, t2, loop_time, time_left, time_limit, quit_time = 1l;
  char *time_limit_string;

  /* Check the clock at the beginning of the run */
  t0 = time(NULL);

  /* Check how much time we have for this job */
  if (! (time_limit_string = getenv ("RSAVE_TIME_LIMIT"))) {
    printf ("Unlimited time for this job.\n");
    timing = FALSE;
  }
  else {
    if (! (0 < sscanf (time_limit_string, "%d", &time_limit))) {
      fprintf (stderr, "Error: bad format of RSAVE_TIME_LIMIT\n");
      exit (1);
    }
    else {
      printf ("Time for this job limited to %d seconds.\n", time_limit);
    }
```

```
}

/* Is this a continued job or a new one? */

if (! (restart = getenv ("RSAVE_RESTART"))) {
  printf ("Starting a new run.\n");
  n = 0;
}
else {
  if (! (restart_name = getenv ("RSAVE_CHECKFILE"))) {
    fprintf (stderr, "error: no checkpoint file for the restart job\n");
    exit (1);
  }
  else {
    printf ("Restarting the job from %s.\n", restart_name);
    if (! (restart_file = fopen(restart_name, "r"))) {
      perror (restart_name);
      exit (2);
    }
    else {
      if (! (fscanf (restart_file, "%d", &n) > 0)) {
        fprintf (stderr, "%s: input file format error\n", restart_name);
        exit (3);
      }
      else {
        fclose (restart_file);
      }
    }
  }
}

printf ("n = %d\n", n);
printf ("\tcomputing ... \n"); fflush (stdout);

/* Loop while keeping an eye on the clock */

do {
  if (timing) t1 = time(NULL);

  sleep (5);
  n++;

  /* Check if the whole simulation has been finished:
     this is our ``convergence'' criterion.
     */
  if (n > LAST_N) finished = TRUE;

  /* Check if we still have enough time for the next loop.
     */
  if (timing) {
     t2 = time(NULL);
     loop_time = t2 - t1;
     time_left = time_limit - (t2 - t0);
     quit_time = time_left - loop_time - SAFETY_MARGIN;
     printf ("\t\tn = %d, time left = %d seconds\n", n, time_left);
     if ((quit_time <= 0) && (! finished))
       printf ("\t\tRun out of time, exiting ... \n");
    }
  } while ((quit_time > 0) && (! finished));

  printf ("\tdone.\n");
  printf ("n = %d\n", n);

  if (! (restart_name = getenv ("RSAVE_CHECKFILE"))) {
    printf ("checkpointing not requested, exiting...\n");
    exit (0);
  }
  else {
    if (restart) {
      strcpy (old_restart_name, restart_name);
      strcat (old_restart_name, ".old");
      printf ("renaming old restart file to %s\n", old_restart_name);
      if (0 > rename (restart_name, old_restart_name)) {
        perror (old_restart_name);
        exit (4);
      }
    }
    printf ("saving data on %s\n", restart_name);
    if (! (restart_file = fopen (restart_name, "w"))) {
      perror (restart_name);
      exit (5);
    }
    else {
      fprintf (restart_file, "%d\n", n);
      fclose (restart_file);
    }
    if (! finished)
      printf ("CONTINUE\n");
    else
      printf ("FINISHED\n");
  }
  exit (0);
}
```

Here is how this job is run. First I submit it with the environmental variable `RSAVE_RESTART` unset, which initialises the job. Then I set `RSAVE_RESTART` to `yes` and resubmit the job, which restarts from where it left.

The job is allowed to run no longer than 30 seconds at a time. Given the safety margin of 10 seconds and a single iteration time of 5 seconds this should let our program do 4 iterations. But the while clause tests for `quit_time > 0` not for `quit_time >= 0`, so, in effect we end up with 3 iterations instead of 4.

While the computational task remains unfinished, program `rts` writes `CONTINUE` on standard output before it exits. But the last run, when `n` becomes 31, is flagged with the word `FINISHED`.

```
gustav@sp19:../LoadLeveler 14:33:15 !556 $ gcc -o rts rts.c
gustav@sp19:../LoadLeveler 14:33:25 !557 $ env | grep RSAVE
RSAVE_TIME_LIMIT=30
RSAVE_CHECKFILE=rts.dat
RSAVE_RESTART=yes
gustav@sp19:../LoadLeveler 14:33:29 !558 $ unset RSAVE_RESTART
```

```
gustav@sp19:../LoadLeveler 14:33:36 !559 $ ./rts
Time for this job limited to 30 seconds.
Starting a new run.
n = 0
        computing ...
                n = 1, time left = 25 seconds
                n = 2, time left = 20 seconds
                n = 3, time left = 15 seconds
                Run out of time, exiting ...
        done.
n = 3
saving data on rts.dat
CONTINUE
gustav@sp19:../LoadLeveler 14:33:53 !560 $ export RSAVE_RESTART="yes"
gustav@sp19:../LoadLeveler 14:34:05 !561 $ ./rts
Time for this job limited to 30 seconds.
Restarting the job from rts.dat.
n = 3
        computing ...
                n = 4, time left = 25 seconds
                n = 5, time left = 20 seconds
                n = 6, time left = 15 seconds
                Run out of time, exiting ...
        done.
n = 6
renaming old restart file to rts.dat.old
saving data on rts.dat
CONTINUE
gustav@sp19:../LoadLeveler 14:34:25 !562 $

...

gustav@sp19:../LoadLeveler 14:38:19 !569 $ ./rts
Time for this job limited to 30 seconds.
Restarting the job from rts.dat.
n = 27
        computing ...
                n = 28, time left = 25 seconds
                n = 29, time left = 20 seconds
                n = 30, time left = 15 seconds
                Run out of time, exiting ...
        done.
n = 30
renaming old restart file to rts.dat.old
saving data on rts.dat
CONTINUE
gustav@sp19:../LoadLeveler 14:38:47 !570 $ ./rts
Time for this job limited to 30 seconds.
Restarting the job from rts.dat.
n = 30
        computing ...
                n = 31, time left = 25 seconds
        done.
n = 31
renaming old restart file to rts.dat.old
saving data on rts.dat
FINISHED
```

```
gustav@sp19:../LoadLeveler 14:39:11 !571 $
```

The Complete Application in Fortran 90

Below is the same code written in Fortran-90.

The wall-clock time is measured using the intrinsic subroutine `system_clock`. As I have already remarked, the number of ticks returned by this subroutine is reset to 0 every midnight. In order to avoid a catastrophe at midnight, we save the clock value returned by the first call to `system_clock` in `clock0`. On all consecutive calls we allways check if the returned value is less than `clock0`, which it will be if the clock has reset in the meantime. If we observe such an event, we add `clock_max` to clock and use the result in our computations. Assuming that your job will not block the queue for more than 24 hours that should work just fine, otherwise additional day counters would have to be included in the logic of the program.

This is basically the only difference between our C and our Fortran-90 versions of the program.

I have made a more extensive use of the `cpp` preprocessor in this code. All major constants have been defined using the `#ifndef .. #endif` clauses at the beginning of the listing. This way their values can be altered from the command line, using the `-D` switch, while generating the Fortran-90 code with `gcc -E -P -C`.

The logic of the `do` loop differs slightly from the logic of the `do` loop in C, because the `while` condition is tested at the beginning of the loop. However, the default values of `quit_time` and `finished` are such that the loop will be always executed at least once. So, in effect, things should work here exactly as in our C example. The initialisation of `quit_time` to 1 also ensures that if timing has not been requested by the user, the job will continue running, until finished. The default values of `timing`, and `finished` have the same effect as in the C example.

```
#ifndef STDOUT
# define STDOUT 6
#endif

#ifndef RESTART_FILE
# define RESTART_FILE 21
#endif

#ifndef LAST_N
# define LAST_N 30
#endif

#ifndef SAFETY_MARGIN
# define SAFETY_MARGIN 10
#endif

#ifndef SHORT_STRING_LEN
# define SHORT_STRING_LEN 64
```

```
#endif

#ifndef LONG_STRING_LEN
# define LONG_STRING_LEN 512
#endif

program rts

  ! R)estore T)ime S)ave

#ifdef XLF
  use xlfutility
#endif
  character (len=SHORT_STRING_LEN) restart, restart_name, old_restart_name
  character (len=LONG_STRING_LEN) command
  integer n, restart_file, status
  parameter (restart_file = RESTART_FILE)

  ! Variables for timing

  integer t0, t1, t2, loop_time, time_left, time_limit, quit_time, &
       count0, count, count_rate, count_max, safety_margin
  character (len=SHORT_STRING_LEN) time_limit_string
  logical timing
  data quit_time /1/, timing /.true./, safety_margin /SAFETY_MARGIN/

  ! Variables for finishing the task

  integer last_n
  logical finished
  data finished /.false./, last_n /LAST_N/

  ! Look up the clock at the beginning of the run

  call system_clock (count0, count_rate, count_max)
  t0 = count0 / count_rate

  ! Check how much time we have for this job

  call getenv ('RSAVE_TIME_LIMIT', time_limit_string)
  if (len_trim (time_limit_string) .eq. 0) then
     write (STDOUT, '(1x, 1a)') 'Unlimited time for this job.'
     timing = .false.
  else
     read (time_limit_string, '(1i7)', iostat=status, err=130, end=130) &
          time_limit
     write (STDOUT, '(1x, 1a, 1i7, 1a)') 'Time for this job limited to ', &
          time_limit, ' seconds'
  end if

  ! Is this a continued job or a new one?

  call getenv ('RSAVE_RESTART', restart)
  if (len_trim(restart) .eq. 0) then
     write (STDOUT, '(1x, 1a)') 'Starting a new run'
     n = 0
  else
```

```
     call getenv ('RSAVE_CHECKFILE', restart_name)
     if (len_trim(restart_name) .eq. 0) then
        write (STDOUT, '(1x, 1a)') &
             'Error: no checkpoint file for the restart job'
        stop 1
     else
        write (STDOUT, '(1x, 2a)') 'Restarting the job from ', restart_name
        open (unit=restart_file, iostat=status, err=100, file=restart_name, &
             status='old', action='read')
        read (restart_file, '(1i7)', iostat=status, err=110, end=110) n
        close (restart_file)
     end if
  end if

  ! This is our computation part

  write (STDOUT, '(1x, 1a, 1i7)') 'n = ', n
  write (STDOUT, '(9x, 1a)') 'computing ... '
#ifdef XLF
  call flush_ (STDOUT)
#endif

  do while ((quit_time .gt. 0) .and. (.not. finished))
     if (timing) then
        call system_clock(count=count)
        if (count .lt. count0) count = count + count_max
        t1 = count / count_rate
     end if

#ifdef XLF
     call sleep_ (5)
#endif
     n = n + 1

     ! Check if the whole simulation has been finished:
     ! this is our ``convergence'' criterion

     if (n > last_n) finished = .true.

     ! Check if we still have enough time for the next loop

     if (timing) then
        call system_clock(count=count)
        if (count .lt. count0) count = count + count_max
        t2 = count / count_rate
        loop_time = t2 - t1
        time_left = time_limit - (t2 - t0)
        quit_time = time_left - loop_time - safety_margin
        write (STDOUT, '(16x, 1a, 1i7, 1a, 1i7, 1a)') &
             'n = ', n, ' time left = ', time_left, ' seconds'
        if ((quit_time .le. 0) .and. (.not. finished)) &
             write (STDOUT, '(16x, 1a)') 'Run out of time, exiting ... '
     end if
  end do

  write (STDOUT, '(9x, 1a)') 'done.'
  write (STDOUT, '(1x, 1a, 1i7)') 'n = ', n
  ! And now we save the result on a new checkpoint file, saving
  ! the old one under a new name if need be.

  call getenv ('RSAVE_CHECKFILE', restart_name)
  if (len_trim(restart_name) .eq. 0) then
     write (STDOUT, '(1x, 1a)') 'Checkpointing not requested, exiting ... '
     stop 0
  else
     if (.not. (len_trim(restart) .eq. 0)) then
        old_restart_name = restart_name (1:len_trim(restart_name)) // '.old'
        write (STDOUT, '(1x, 2a)') 'Renaming the old restart file to ', &
             old_restart_name
        command = 'mv' // ' ' // restart_name // ' ' // old_restart_name
        call system (command)
     end if
     write (STDOUT, '(1x, 2a)') 'Saving data on ', restart_name
     open (unit=restart_file, iostat=status, err=120, file=restart_name, &
          status='replace', action='write')
     write (restart_file, '(1i7)') n
     close (restart_file)
     if (.not. finished) then
        write (STDOUT, '(1x, 1a)') 'CONTINUE'
     else
        write (STDOUT, '(1x, 1a)') 'FINISHED'
     end if
  end if
  stop 0

  ! error handlers

  ! error while opening the checkpoint file for reading

100 write (STDOUT, '(1x, 3a)') 'Error: while opening ', restart_name, &
       ' for reading'
  write (STDOUT, '(8x, 1a, 1i7)') 'iostat = ', status
  stop 2

  ! error while trying to read input file

110 write (STDOUT, '(1x, 2a)') 'Error: while reading from ', restart_name
  write (STDOUT, '(8x, 1a, 1i7)') 'iostat = ', status
  stop 3

  ! error while opening the checkpoint file for writing

120 write (STDOUT, '(1x, 3a)') 'Error: while opening ', restart_name, &
       ' for writing'
  write (STDOUT, '(8x, 1a, 1i7)') 'iostat = ', status
  stop 5

  ! error while trying to read from time_limit_string

130 write (STDOUT, '(1x, 1a)') 'Error: bad format of RSAVE_TIME_LIMIT'
  write (STDOUT, '(8x, 1a, 1i7)') 'iostat = ', status
  stop 6
```

```
end program rts
```

The program can be compiled as follows:

```
gustav@sp19:../LoadLeveler 14:45:36 !573 $ gcc -E -P -C -DXLF rts.cpp > rts.f
gustav@sp19:../LoadLeveler 14:46:01 !574 $ xlf90 -o rts rts.f
** rts   === End of Compilation 1 ===
1501-510  Compilation successful for file rts.f.
gustav@sp19:../LoadLeveler 14:46:07 !575 $
```

And here is how I've run it. Observe another subtle difference between our C and Fortran-90 examples: when the Fortran program exits, apart from writing CONTINUE or FINISHED it also writes STOP 0. If our LoadLeveler script was to inspect only the last line of the log file for the word CONTINUE, we would have missed it in this case. So, instead, the script will grep through the whole file. Of course, this assumes that a new log file will be created each time.

```
gustav@sp19:../LoadLeveler 14:46:19 !576 $ env | grep RSAVE
RSAVE_TIME_LIMIT=30
RSAVE_CHECKFILE=rts.dat
RSAVE_RESTART=yes
gustav@sp19:../LoadLeveler 14:46:36 !577 $ unset RSAVE_RESTART
gustav@sp19:../LoadLeveler 14:46:42 !578 $ ./rts
 Time for this job limited to      30 seconds
 Starting a new run
 n =       0
        computing ...
              n =       1 time left =      25 seconds
              n =       2 time left =      20 seconds
              n =       3 time left =      15 seconds
              Run out of time, exiting ...
        done.
 n =       3
 Saving data on rts.dat
 CONTINUE
STOP 0
gustav@sp19:../LoadLeveler 14:47:01 !579 $ export RSAVE_RESTART="yes"
gustav@sp19:../LoadLeveler 14:47:24 !580 $ ./rts
 Time for this job limited to      30 seconds
 Restarting the job from rts.dat
 n =       3
        computing ...
              n =       4 time left =      25 seconds
              n =       5 time left =      20 seconds
              n =       6 time left =      15 seconds
              Run out of time, exiting ...
        done.
 n =       6
 Renaming the old restart file to rts.dat.old
 Saving data on rts.dat
 CONTINUE
STOP 0
gustav@sp19:../LoadLeveler 14:47:42 !581 $
```

```
...

gustav@sp19:../LoadLeveler 14:52:46 !588 $ ./rts
 Time for this job limited to      30 seconds
 Restarting the job from rts.dat
 n =      27
        computing ...
              n =      28 time left =      25 seconds
              n =      29 time left =      20 seconds
              n =      30 time left =      15 seconds
              Run out of time, exiting ...
        done.
 n =      30
 Renaming the old restart file to rts.dat.old
 Saving data on rts.dat
 CONTINUE
STOP 0
gustav@sp19:../LoadLeveler 14:53:15 !589 $ ./rts
 Time for this job limited to      30 seconds
 Restarting the job from rts.dat
 n =      30
        computing ...
              n =      31 time left =      25 seconds
        done.
 n =      31
 Renaming the old restart file to rts.dat.old
 Saving data on rts.dat
 FINISHED
STOP 0
gustav@sp19:../LoadLeveler 14:53:31 !590 $
```

7.5.4 Combining the Application with LoadLeveler: Automatic Resubmission

In this section I shall demonstrate how our toy application can be run under the LoadLeveler, and how you can use its various features to automatically keep resubmitting the job until the whole computational task is finished.

What makes it particularly easy is the LoadLeveler's

```
#@environment=COPY_ALL
```

statement, which transfers all currently defined environmental variables to the submitted job. That way we can define, say, `RSAVE_RESTART` in the script, after the first, initialising run of the application, and rest assured that when the job is resubmitted, it will already read the data from the restart file.

The LoadLeveler script begins by running program `./rts`: that is our application. The output is saved on `rts.log`:

```
./rts > rts.log
```

Both C and Fortran examples are invoked in the same way.

After the job exits the script performs a number of quite interesting manipulations. First of all, it checks if an environmental variable RSAVE_STEP exists. That variable is used to number our LoadLeveler runs. It is quite like LoadLeveler's variable $(stepid), with the difference that here we do it all ourselves. If the variable exists, it means that this particular run was already a resubmission. In that case the value of RSAVE_STEP is incremented and the old restart file, say, rts.dat.old is renamed to something like rts.dat.3,, where 3 is the RSAVE_STEP number. That way we keep the log of the whole computation. In a more complex application, the rts.dat files could contain images or three dimensional data sets, which, if saved, could be used to produce an animation or a CAVE display.

If the variable RSAVE_STEP does not exist, it means that this is the initialising run. In that case the variable is created and assigned number 0. Because we export it, it will become available to the next instantiation of the job.

The log file, rts.log is also saved on something like, say, rts.log.3, where 3 is the RSAVE_STEP number. Observe that rts.log.3 corresponds to the run that used rts.dat.3 as its restart file.

After these manipulations we inspect the log file itself and check if it contains the word CONTINUE. If it does, we check if the variable RSAVE_RESTART exists. If it doesn't, it means that this was the first, initialising run. So we create that variable. Once created it will become available to the next instantiation of the job via the #@environment=COPY_ALL mechanism. Either way the job is resubmitted with the command

```
llsubmit $LOADL_STEP_COMMAND
```

where $LOADL_STEP_COMMAND evaluates to the name of the LoadLeveler script itself.

If the word CONTINUE has not been found in the log file, then we check if the log file contains the word FINISHED. If the job is FINISHED it is not resubmitted. Instead a mail message is sent to whoever submitted the job in the first place ($LOADL_STEP_OWNER), informing the addressee that the job has been completed.

If neither the word CONTINUE nor the word FINISHED have been found in the log file, it means that an error condition must have occurred and the job exited mid-way. In that case, the job is not resubmitted and a mail message informing about the error is sent to the $LOADL_STEP_OWNER.

Here is the whole LoadLeveler script in full glory:

```
# @ shell = /afs/ovpit.indiana.edu/@sys/gnu/bin/bash
# @ environment = COPY_ALL
# @ job_name = rts
# @ output = $(job_name).$(jobid).out
# @ error = $(job_name).$(jobid).err
# @ class = test
# @ notification = never
# @ queue
#
# Execute this step.
#
./rts > rts.log
```

```
#
# If there is $RSAVE_CHECKFILE.old file then
# replace the suffix ".old" with a step number.
#
if [ -n "${RSAVE_STEP}" ]
then
   export RSAVE_STEP=`expr $RSAVE_STEP + 1`
   if [ -n "${RSAVE_CHECKFILE}" ]
   then
      if [ -f $RSAVE_CHECKFILE.old ]
      then
         mv $RSAVE_CHECKFILE.old $RSAVE_CHECKFILE.$RSAVE_STEP
      fi
   fi
else
   export RSAVE_STEP=0
fi
#
# also save the log of this run
#
cp rts.log rts.log.$RSAVE_STEP
#
# Check if the job is finished and if it is not
# resubmit this file
#
if grep CONTINUE rts.log
then
   if [ -z "${RSAVE_RESTART}" ]
   then
      export RSAVE_RESTART=yes
   fi
   llsubmit $LOADL_STEP_COMMAND
elif grep FINISHED rts.log
then
   mailx $LOADL_STEP_OWNER << EOF
Your job rts has FINISHED
EOF
else
   mailx $LOADL_STEP_OWNER << EOF
rts: error exit, check the log file
EOF
fi
```

Here is how this script is submitted and what happens afterwards.

```
gustav@sp19:../LoadLeveler 15:09:24 !620 $ env | grep RSAVE
RSAVE_TIME_LIMIT=30
RSAVE_CHECKFILE=rts.dat
gustav@sp19:../LoadLeveler 15:09:35 !621 $ llsubmit rts.ll
submit: The job "sp19.104" has been submitted.
gustav@sp19:../LoadLeveler 15:09:40 !622 $
```

Observe that only RSAVE_TIME_LIMIT and RSAVE_CHECKFILE have been defined. All other variables will be defined by the LoadLeveler script as they become needed.

The job runs happily resubmitting itself every time the program `rts` exits and producing numerous log and data files:

```
gustav@sp19:../LoadLeveler 15:27:09 !695 $ ls rts*
rts          rts.169.err  rts.448.out  rts.dat.10   rts.f        rts.log.5
rts.105.err  rts.169.out  rts.449.err  rts.dat.2    rts.ll       rts.log.6
rts.105.out  rts.445.err  rts.449.out  rts.dat.3    rts.log      rts.log.7
rts.166.err  rts.445.out  rts.98.err   rts.dat.4    rts.log.0    rts.log.8
rts.166.out  rts.446.err  rts.98.out   rts.dat.5    rts.log.1    rts.log.9
rts.167.err  rts.446.out  rts.c        rts.dat.6    rts.log.10
rts.167.out  rts.447.err  rts.cpp      rts.dat.7    rts.log.2
rts.168.err  rts.447.out  rts.dat      rts.dat.8    rts.log.3
rts.168.out  rts.448.err  rts.dat.1    rts.dat.9    rts.log.4
gustav@sp19:../LoadLeveler 15:30:13 !696 $
```

The `rts.dat.*` files contain the evolution (or *animation*) of the system:

```
 gustav@sp19:../LoadLeveler 15:30:13 !696 $ cat `ls -t rts.dat.*`
     30
     27
     24
     21
     18
     15
     12
      9
      6
      3
gustav@sp19:../LoadLeveler 15:31:22 !697 $
```

The `rts.log.*` files contain the log of the whole computation:

```
gustav@sp19:../LoadLeveler 15:31:22 !697 $ cat `ls -t rts.log.*`
 Time for this job limited to      30 seconds
 Restarting the job from rts.dat
 n =      30
         computing ...
                n =      31 time left =      25 seconds
         done.
 n =      31
 Renaming the old restart file to rts.dat.old
 Saving data on rts.dat
 FINISHED
 Time for this job limited to      30 seconds
 Restarting the job from rts.dat
 n =      27
         computing ...
                n =      28 time left =      25 seconds
                n =      29 time left =      20 seconds
                n =      30 time left =      15 seconds
                Run out of time, exiting ...
         done.
 n =      30
 Renaming the old restart file to rts.dat.old
 Saving data on rts.dat
 CONTINUE
```

```
...

 Time for this job limited to      30 seconds
 Restarting the job from rts.dat
 n =       3
        computing ...
              n =       4 time left =      25 seconds
              n =       5 time left =      20 seconds
              n =       6 time left =      15 seconds
              Run out of time, exiting ...
        done.
 n =       6
 Renaming the old restart file to rts.dat.old
 Saving data on rts.dat
 CONTINUE
 Time for this job limited to      30 seconds
 Starting a new run
 n =       0
        computing ...
              n =       1 time left =      25 seconds
              n =       2 time left =      20 seconds
              n =       3 time left =      15 seconds
              Run out of time, exiting ...
        done.
 n =       3
 Saving data on rts.dat
 CONTINUE
gustav@sp19:../LoadLeveler 15:32:16 !698 $
```

And the `rts.*.out` files contain messages from the LoadLeveler script in its various instantiations:

```
gustav@sp19:../LoadLeveler 15:32:16 !698 $ cat `ls -t rts.*.out`
 FINISHED
 CONTINUE
submit: The job "sp18.169" has been submitted.
 CONTINUE
submit: The job "sp17.449" has been submitted.
 CONTINUE
submit: The job "sp18.168" has been submitted.
 CONTINUE
submit: The job "sp17.448" has been submitted.
 CONTINUE
submit: The job "sp18.167" has been submitted.
 CONTINUE
submit: The job "sp17.447" has been submitted.
 CONTINUE
submit: The job "sp18.166" has been submitted.
 CONTINUE
submit: The job "sp17.446" has been submitted.
 CONTINUE
submit: The job "sp21.98" has been submitted.
 CONTINUE
submit: The job "sp17.445" has been submitted.
```

```
gustav@sp19:../LoadLeveler 15:34:36 !699 $
```

When the whole job finished I have received the following mail message sent to me by the LoadLeveler script:

```
Date: Tue, 26 Jan 1999 15:26:56 -0500
From: Zdzislaw Meglicki <gustav@sp17.ucs.indiana.edu>
Message-Id: <199901262026.PAA18102@sp17.ucs.indiana.edu>
To: gustav@sp17.ucs.indiana.edu
Content-Type: text
Content-Length: 26

Your job rts has FINISHED
```

If you run a long job, which resubmits itself twice or perhaps only once a day, it is a good idea to change

```
#@notification = never
```

to

```
#@notification = always
```

so that you can keep an eye on the computation.

Chapter 8

Message Passing Interface

8.1 Introduction

8.1.1 The History of MPI

- The progenitors:
  - PICL, PVM – Oak Ridge National Laboratory
  - PARMACS, P4, Chameleon – Argonne National Laboratory
  - Express – Caltech/ParaSoft
  - LAM – Ohio Supercomputer Center
  - TCGMSG – Specially designed for Quantum Chemistry
- One day (April 1992, during one-day workshop on Standards for Message Passing in Distributed-Memory Environment) they all realised that they were continuously reinventing the wheel duplicating each other's efforts
- They got together (Supercomputing '92) and decided to thrash out a common standard in the same hotel in Dallas, where HPF Forum met
- Some well known groups, e.g., ISIS (Cornell University), Linda (Yale), stayed away from the initiative, to their peril
- From the moment of conception, like P4 and Express, MPI was unashamedly biased towards HPC and a static process pool
- Industrial participants included Convex, Cray, IBM, Intel, Meiko, nCUBE, NEC, Thinking Machines
- The first standard (MPI-1.0) was completed in May 1994
- The second, enhanced standard (MPI-2.0) is being completed now

8.1.2 MPI Literature

- MIT Press Books
  - "MPI: The Complete Reference", by Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra, November 1995, ISBN 0-262-69184-1, 336 pages, $US27.50 (paper)
    * `http://mitpress.mit.edu/mitp/recent-books/comp/snimp.html`
  - "Using MPI", by by William Gropp, Ewing Lusk, and Anthony Skjellum, 1994, ISBN 0-262-57104-8, 328 pages, $US28.50 (paper)
    * `http://mitpress.mit.edu/mitp/recent-books/comp/group.html`
  - "Parallel Programming Using C++", edited by Gregory V. Wilson and Paul Lu, July 1996, ISBN 0-262-73118-5, 760 pages, $US45.00 (paper)
    * `http://mitpress.mit.edu/mitp/recent-books/comp/wilrp.html`
- On-Line materials
  - "MPI: A Message-Passing Interface Standard, June 12, 1995 (HTML)"
  - "MPI: A Message-Passing Interface Standard, June 12, 1995 (PostScript)"
  - "MPI-2: Extensions to the Message-Passing Interface, November 7, 1996"
  - The Message Passing Interface (MPI) Standard – Main WWW Page
  - ROMIO: A High-Performance, Portable MPI-IO Implementation

8.1.3 What Is New and Old about MPI?

Well, now, nothing really: MPI has been around since 1994, and 5 years in computing is an epoch. And even when it was originally released, it was based on ideas that were around since early 80s. However, in its day MPI capitalised on what people knew about message passing programming already and offered:

- efficiency, portability, and functionality without compromise
- `MPI_Send(buffer, count, datatype, destination, tag, communicator)`
  - `datatype` and `communicator` are new
  - MPI data type are closely related to, but not necessarily identical with UNIX/C/Fortran data types
- rich set of collective communications
- virtual topologies (Express had those)
- hooks for debugging and profiling
- blocking and non-blocking sends and receives

- support for libraries
- support for heterogeneous networks (PVM had that)

At roughly the same time people played also with parallel programming languages (and still do) and with other parallel programming concepts, e.g., shared memory, NUMA shared memory, simulated shared memory, parallel objects, CORBA, ISIS, RPCs, and more.

MPI was a conservative endeavour:

- It's a library, not a language
- It implements the message-passing model

8.1.4 What Is Missing in MPI-1?

- Dynamic process creation (PVM and ISIS had that), but it comes back in MPI-2.
- No portable I/O model, but it comes in MPI-2 (thanks to NASA).
- Virtual Synchrony (ISIS had that) – we probably won't see that in MPI, but we get it back in Microsoft's Wolfpack.

Dynamic process creation is very useful for a broader range of applications, especially applications which must be fault tolerant (e.g., factory floor, stock exchange, hospitals, space shuttle), but is of lesser importance for scientific and engineering *computational* programs.

8.1.5 Size of MPI

MPI is *both* small and large.

- A minimal set of 6 functions is sufficient to begin writing MPI programs:

`MPI_Init`	Initialise MPI
`MPI_Comm_size`	Find out how many processes there are
`MPI_Comm_rank`	Find out which process I am
`MPI_Send`	Send a message
`MPI_Recv`	Receive a message
`MPI_FINALIZE`	Terminate MPI

But already MPI-1 offers a great wealth of functions and MPI-2 offers many more.

Here are all MPI-1 functions listed in alphabetical order:

MPI_ABORT	MPI_ADDRESS	MPI_ALLGATHER	MPI_ALLGATHERV
MPI_ALLREDUCE	MPI_ALLTOALL	MPI_ALLTOALLV	MPI_ATTR_DELETE
MPI_ATTR_GET	MPI_ATTR_PUT	MPI_BARRIER	MPI_BCAST
MPI_BSEND	MPI_BSEND_INIT	MPI_BUFFER_ATTACH	MPI_BUFFER_DETACH
MPI_CANCEL	MPI_CARTDIM_GET	MPI_CART_COORDS	MPI_CART_CREATE
MPI_CART_GET	MPI_CART_MAP	MPI_CART_RANK	MPI_CART_SHIFT
MPI_CART_SUB	MPI_COMM_COMPARE	MPI_COMM_CREATE	MPI_COMM_DUP
MPI_COMM_FREE	MPI_COMM_GROUP	MPI_COMM_RANK	MPI_COMM_REMOTE_GROUP
MPI_COMM_REMOTE_SIZE	MPI_COMM_SIZE	MPI_COMM_SPLIT	MPI_COMM_TEST_INTER
MPI_DIMS_CREATE	MPI_ERRHANDLER_CREATE	MPI_ERRHANDLER_FREE	MPI_ERRHANDLER_GET
MPI_ERRHANDLER_SET	MPI_ERROR_CLASS	MPI_ERROR_STRING	MPI_FINALIZE
MPI_GATHER	MPI_GATHERV	MPI_GET_COUNT	MPI_GET_ELEMENTS
MPI_GET_PROCESSOR_NAME	MPI_GRAPHDIMS_GET	MPI_GRAPH_CREATE	MPI_GRAPH_GET
MPI_GRAPH_MAP	MPI_GRAPH_NEIGHBORS	MPI_GRAPH_NEIGHBORS_COUNT	MPI_GROUP_COMPARE
MPI_GROUP_DIFFERENCE	MPI_GROUP_EXCL	MPI_GROUP_FREE	MPI_GROUP_INCL
MPI_GROUP_INTERSECTION	MPI_GROUP_RANGE_EXCL	MPI_GROUP_RANGE_INCL	MPI_GROUP_RANK
MPI_GROUP_SIZE	MPI_GROUP_TRANSLATE_RANKS	MPI_GROUP_UNION	MPI_IBSEND
MPI_INIT	MPI_INITIALIZED	MPI_INTERCOMM_CREATE	MPI_INTERCOMM_MERGE
MPI_IPROBE	MPI_IRECV	MPI_IRSEND	MPI_ISEND
MPI_ISSEND	MPI_KEYVAL_CREATE	MPI_KEYVAL_FREE	MPI_OP_CREATE
MPI_OP_FREE	MPI_PACK	MPI_PACK_SIZE	MPI_PCONTROL
MPI_PROBE	MPI_RECV	MPI_RECV_INIT	MPI_REDUCE
MPI_REDUCE_SCATTER	MPI_REQUEST_FREE	MPI_RSEND	MPI_RSEND_INIT
MPI_SCAN	MPI_SCATTER	MPI_SCATTERV	MPI_SEND
MPI_SENDRECV	MPI_SENDRECV_REPLACE	MPI_SEND_INIT	MPI_SSEND
MPI_SSEND_INIT	MPI_START	MPI_STARTALL	MPI_TEST
MPI_TESTALL	MPI_TESTANY	MPI_TESTSOME	MPI_TEST_CANCELLED
MPI_TOPO_TEST	MPI_TYPE_COMMIT	MPI_TYPE_CONTIGUOUS	MPI_TYPE_EXTENT
MPI_TYPE_FREE	MPI_TYPE_HINDEXED	MPI_TYPE_HVECTOR	MPI_TYPE_INDEXED
MPI_TYPE_LB	MPI_TYPE_SIZE	MPI_TYPE_STRUCT	MPI_TYPE_UB
MPI_TYPE_VECTOR	MPI_UNPACK	MPI_WAIT	MPI_WAITALL
MPI_WAITANY	MPI_WAITSOME	MPI_WTICK	MPI_WTIME

8.1.6 MPI Examples

`ftp.mcs.anl.gov:/pub/mpi/using/examples/examples.tar.gz`

These are examples from the book "Using MPI" by Gropp, Lusk, and Skjellum.

8.2 Simple MPI

8.2.1 Hello World

```
#include <stdio.h>
#include <mpi.h>

main(argc, argv)
int argc;
char *argv[];
{
        char name[BUFSIZ];
        int length;

        MPI_Init(&argc, &argv);
        MPI_Get_processor_name(name, &length);
        printf("%s: hello world\n", name);
        MPI_Finalize();
}
```

This is a program we have already seen before, when we talked about running MPI programs under LoadLeveler.

There is an MPI wrapper on the SP that takes care of includes and libraries. Compile and link this program in one step with:

```
gustav@sp20:../LoadLeveler 17:07:04 !513 $ mpcc mpi-hello.c -o mpi-hello
```

```
gustav@sp20:../LoadLeveler 17:07:23 !514 $
```

and run it by submitting the following LoadLeveler script:

```
gustav@sp20:../LoadLeveler 19:27:47 !647 $ cat mpi-hello.ll
# @ job_type = parallel
# @ environment = COPY_ALL; MP_EUILIB=ip; MP_INFOLEVEL=3
# @ requirements = (Adapter == "hps_ip")
# @ min_processors = 4
# @ max_processors = 8
# @ class = test
# @ notification = always
# @ executable = /usr/bin/poe
# @ arguments = mpi-hello
# @ output = mpi-hello.out
# @ error = mpi-hello.err
# @ queue
gustav@sp20:../LoadLeveler 19:27:49 !648 $ llsubmit mpi-hello.ll
```

If you want or need to exclude certain nodes from your processor pool add the following to the `requirements` directive:

```
( Machine != "sp18" ) && ( Machine != "sp20" )
```

When the job completes you should see something like:

```
gustav@sp20:../LoadLeveler 19:27:49 !648 $ cat mpi-hello.out
sp21.ucs.indiana.edu: hello world
sp19.ucs.indiana.edu: hello world
sp24.ucs.indiana.edu: hello world
sp20.ucs.indiana.edu: hello world
sp22.ucs.indiana.edu: hello world
sp23.ucs.indiana.edu: hello world
sp17.ucs.indiana.edu: hello world
sp18.ucs.indiana.edu: hello world
gustav@sp20:../LoadLeveler 19:29:11 !649 $
```

on your log file.

All MPI programs must begin with `MPI_Init(&argc, &argv)` and end with `MPI_Finalize()`. It is not an error to insert C or Fortran statements in front of `MPI_Init` or after `MPI_Finalize`, but MPI standard is not concerned with how such statements should be executed, if at all, e.g., on one processor, or on all of them. In short if you write a program like that, it will be unpredictable and non-portable. So, don't do it.

All MPI functions in C interface begin with `MPI_X`, where `X` stands for a capital letter that begins the proper name of the function, e.g., `MPI_Get_processor_name`. The latter is one of the functions from the chapter about *Environmental Enquiries*.

You seldom need to be concerned about the name of the processor your MPI process runs on. The reason why MPI Founding Fathers decided on this function at all is to allow for process migration. The idea is that a program may distribute itself over a number of workstations. If anyone of those workstations is requested back by its "owner", your parallel program will migrate the process that runs on it elsewhere. Then it may keep checking occasionally if the workstation is again available, and if it is, the process will be moved back.

In our short example, we use this function simply to demonstrate that the program indeed runs on multiple CPUs.

The `printf` statement assumes that all processes comprising an MPI program have access to standard output on "MPI console", that is your VDU, if you run the program interactively, or a file that LoadLeveler is going to write standard output on. This assumption may or may not be satisfied by the hardware and software your MPI program runs on. It is not a requirement of MPI standard.

There are systems where only some processes can do any IO at all, and sometimes only one process can do standard IO.

MPI provides means to check for that. By calling function

```
MPI_Attr_get
```

you can inspect the value of various MPI attributes that would have been generated dynamically when the program begins its execution. Amongst these are

MPI_HOST which specifies the rank of the process that runs on a host machine. Some parallel computers must run off a host machine, e.g., the Connection Machine always had to be front-ended by a Sun, or by a VAX, and it was possible to run an MPI job in such a way that one of the processes would run on that front-end machine. That would be the *host process*.

MPI_IO which specifies a rank of a node that has regular I/O. You can use this attribute so that every process can find on its own if it has I/O. Then processes can communicate that amongst themselves, find a group of processes that *support* regular I/O, and redirect all I/O through them.

8.2.2 Greetings, Master

The "Hello World" program from section 8.2.1 ran in parallel, but participating processes did not exchange any messages, so the parallelism was trivial.

In this section we're going to have a look at our first non-trivial parallel program.

Here it is:

```
#include <stdio.h>
#include <string.h>
#include <mpi.h>

#define TRUE 1
#define FALSE 0
#define MASTER_RANK 0
```

```
main(argc, argv)
int argc;
char *argv[];
{
   int count, pool_size, my_rank, my_name_length, i_am_the_master = FALSE;
   char my_name[BUFSIZ], master_name[BUFSIZ], send_buffer[BUFSIZ],
        recv_buffer[BUFSIZ];
   MPI_Status status;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &pool_size);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
   MPI_Get_processor_name(my_name, &my_name_length);

   if (my_rank == MASTER_RANK) {
      i_am_the_master = TRUE;
      strcpy (master_name, my_name);
   }

   MPI_Bcast(master_name, BUFSIZ, MPI_CHAR, MASTER_RANK, MPI_COMM_WORLD);

   sprintf(send_buffer, "hello %s, greetings from %s, rank = %d",
           master_name, my_name, my_rank);
   MPI_Send (send_buffer, strlen(send_buffer) + 1, MPI_CHAR,
             MASTER_RANK, 0, MPI_COMM_WORLD);

   if (i_am_the_master) {
      for (count = 1; count <= pool_size; count++) {
         MPI_Recv (recv_buffer, BUFSIZ, MPI_CHAR, MPI_ANY_SOURCE, MPI_ANY_TAG,
                   MPI_COMM_WORLD, &status);
         printf ("%s\n", recv_buffer);
      }
   }
   MPI_Finalize();
}
```

And here is the synopsis of the program:

1. Each process finds out about the size of the process pool, its own rank within the pool, and the name of the processor it runs on.

2. Process of rank 0 becomes the master process.

3. The master process broadcasts the name of the processor it runs on to other processes.

4. Each process, including the master process constructs a greating message and sends it to the master process. The master process sends the message to itself.

5. The master process collects the messages and displays them on standard output.

6. This is the way to organise I/O, if only certain processes can write to the screen or to files.

Let us compile and run this program on our SP:

```
gustav@sp19:../MPI 20:51:34 !511 $ mpcc -o hello hello.c
gustav@sp19:../MPI 20:52:01 !512 $ cat hello.ll
# @ job_type = parallel
# @ environment = COPY_ALL; MP_EUILIB=ip; MP_INFOLEVEL=3
# @ requirements = (Adapter == "hps_ip") && (Machine != "sp20") \
                   && (Machine != "sp18")
# @ min_processors = 4
# @ max_processors = 8
# @ class = test
# @ notification = never
# @ executable = /usr/bin/poe
# @ arguments = hello
# @ output = hello.out
# @ error = hello.err
# @ queue
gustav@sp19:../MPI 20:52:06 !513 $ llsubmit hello.ll
submit: The job "sp19.106" has been submitted.
gustav@sp19:../MPI 20:52:11 !514 $ cat hello.out
hello sp24.ucs.indiana.edu, greetings from sp24.ucs.indiana.edu, rank = 0
hello sp24.ucs.indiana.edu, greetings from sp23.ucs.indiana.edu, rank = 1
hello sp24.ucs.indiana.edu, greetings from sp22.ucs.indiana.edu, rank = 3
hello sp24.ucs.indiana.edu, greetings from sp21.ucs.indiana.edu, rank = 2
hello sp24.ucs.indiana.edu, greetings from sp17.ucs.indiana.edu, rank = 4
hello sp24.ucs.indiana.edu, greetings from sp19.ucs.indiana.edu, rank = 5
gustav@sp19:../MPI 20:52:45 !515 $
```

Now let us explain in more detail what happens here.

When you look at an MPI program and try to trace its logic, think of yourself as one of the processors.

And so, you begin execution and the first statement that you encounter is

```
MPI_Init(&argc, &argv);
```

What this statement tells you is that *you are not alone*. There are others like you, and all of you comprise a pool of MPI processes. How many there are in that pool altogether? To find out you issue the command

```
MPI_Comm_size(MPI_COMM_WORLD, &pool_size);
```

which, translated into English means:

> *How many processes there are in the default communicator, which is guaranteed to encompass all processes in the pool,* `MPI_COMM_WORLD`*? Please put the answer in the variable* `pool_size`.

When this function returns you know how many colleagues you have. But the next pressing question is: how can you distinguish yourself from the others? Are you all alike? Are you all indistinguishable?

When processes are born, each process is born with a different number, much the same as each human is born with different DNA and different fingerprints. That number is called a *rank number*, and if you are an MPI process you can find out what *your* rank number is by calling function:

```
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
```

The English translation of this call is:

> *What is my rank number in the default communicator* `MPI_COMM_WORLD`*?*
> *Please put the answer in the variable* `my_rank`*.*

A process such as yourself can belong to many communicators. You always belong to `MPI_COMM_WORLD`, but within the world you can have many sub-worlds, or, let's call it *states*. If you have multiple citizenships, you will also have multiple tax numbers, or multiple social security numbers, that would distinguish you from other citizens of those states. By the same token a process that belongs to many communicators may have different a different rank number in each of them, so when you ask about your rank number you must specify a communicator too.

OK, by now you know how many other processes there are in the pool, and what is your rank number within that pool. You can also find the name of the process*or* that you yourself run on, and this is done in a way that you've already seen in section 8.2.1. You call function:

```
MPI_Get_processor_name(my_name, &my_name_length);
```

which translated to English means:

> *What is the name of the processor that I run on? Please put the name in the variable* `my_name` *and put the length of that name in* `my_name_length`*.*

So far every process in the pool would have performed exactly the same operations. There has been no communication between you guys yet. But now you all check if your rank number is the same as a predefined `MASTER_RANK` number. Who defines what the `MASTER_RANK` number is? In this case it is the programmer, the God of MPI processes. But on some systems all processes may go through additional environmental enquiries and check for the existence of a host process or processes which can do I/O, and so on, and then jointly decide on which is going to be the `MASTER`.

Well, here the `MASTER` has been annointed by God.

Only one process will discover that he or she is the annointed one. That one process will place `TRUE` in the `i_am_the_master` variable. For all other processes that variable will remain `FALSE`. This one process will laboriously copy its name into the variable `master_name`. For all other processes that string will remain null.

But all other processes will know that they are not the master, and they will know who the master is, because by now they all know that their rank is *not* `MASTER_RANK`.

At this stage all processes that are not the master subject themselves to receiving a broadcast from the master. All processes, including yourself (regardless of whether you are the master or not), perform this operation at the same time, and all of them end up with the same message in the variable `master_name`. This message is the name of the processor the master process runs on. The name has been copied from the variable `master_name` of the master process and written on variables called `master_name` that belong to other processes. The MPI machine will have done all that.

This operation is accomplished by calling:

```
MPI_Bcast(master_name, BUFSIZ, MPI_CHAR, MASTER_RANK, MPI_COMM_WORLD);
```

In plain English the meaning of this call is as follows:

> *Copy* `BUFSIZ` *data items of type* `MPI_CHAR` *from a buffer called* `master_name` *that is managed by process whose rank is* `MASTER_RANK` *within the* `MPI_COMM_WORLD` *communicator, to which I must belong too, to my own buffer also called* `master_name`*.*

At this stage whether you are a slave process or a master process you are very knowledgeable about your `MPI_COMM_WORLD` universe. And, if you are a slave process, you are prudent enough to prepare and send a congratulatory message to the master process. And so first you write the message on your `send_buffer`:

```
sprintf(send_buffer, "hello %s, greetings from %s, rank = %d",
        master_name, my_name, my_rank);
```

And observe that you write this message even if you are the master. Well there is nothing wrong with congratulating yourself. Some people do it all the time.

Having prepared the message you send it to the master process, and if you are the master process you send it to yourself, which is fine too. Some people seldom receive messages from anyone else.

Here is how you will have accomplished this task:

```
MPI_Send (send_buffer, strlen(send_buffer) + 1, MPI_CHAR,
          MASTER_RANK, 0, MPI_COMM_WORLD);
```

In plain English the meaning of this operation is as follows:

> *Send* `strlen(send_buffer) + 1` *data items (don't forget about the terminating null character, for which function* `strlen` *does not account) of type* `MPI_CHAR`*, which have been deposited in* `send_buffer` *to a process whose rank is* `MASTER_RANK`*. Attach a* tag `0` *to that message (to distinguish it from other messages that the master process may receive from elsewhere, perhaps). The ranking and communication refer to the* `MPI_COMM_WORLD` *communicator.*

If you are a slave process then this is about all that you are supposed to do in this program, so now you can relax and spin, or go home.

But if you are a master process you have to collect all those messages that have been sent to you and print them on standard output in the *receive* order.

How many messages are you going to receive, master? There will be `pool_size` messages sent to you from all processes including yourself. So you can just as well enter a `for` loop and receive all those `pool_size` messages, knowing, when you count the last one, that your job is done too.

To receive a message you do as follows:

```
   MPI_Recv (recv_buffer, BUFSIZ, MPI_CHAR, MPI_ANY_SOURCE, MPI_ANY_TAG,
             MPI_COMM_WORLD, &status);
```

which in plain English means:

> *Let me receive up to* `BUFSIZ` *data items of type* `MPI_CHAR` *into my array* `recv_buffer` *from any source (*`MPI_ANY_SOURCE`*) and with any tag (*`MPI_ANY_TAG`*) within the* `MPI_COMM_WORLD`*. The status of the received message should be written on structure* `status`*.*

It is possible to find out a lot about a message *before* you are going to receive it. You can find how long it is, where it comes from, what type are data items inside the message, and so on. But in this case the master process doesn't bother. The logic of the program is simple enough. God, i.e., the programmer, told the master process to receive `pool_size` messages, so receive them it shall. And it shall it print them on standard output as it receives them.

Once this point in the program is reached, all processes hit `MPI_Finalize`, which is the end of the world for them.

And the beginning of the debugging process for the Programmer.

8.2.3 Dividing the Pie

Sending congratulatory messages is all very well, but it makes few people happy other than the Programmer and the master process.

So here is an example of a very simple program that calculates π. This, at least, should make some high school teachers happy.

```
#include <stdio.h>
#include <mpi.h>
#define FALSE 0
#define TRUE  1
#define MASTER_RANK 0

double f(a)
double a;
{
   return (4.0 / (1.0 + a*a));
}

int main ( int argc, char **argv )
{
   int n, i, pool_size, my_rank, i_am_the_master = FALSE;
   double mypi, pi, h, sum, x, a;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &pool_size);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   if (my_rank == MASTER_RANK) i_am_the_master = TRUE;

   if (i_am_the_master) {
      printf("Enter the number of intervals: ");
      scanf("%d",&n);
      if (n==0) n=100;
   }

   MPI_Bcast(&n, 1, MPI_INT, MASTER_RANK, MPI_COMM_WORLD);

   h   = 1.0 / (double) n;
   sum = 0.0;
   for (i = my_rank + 1; i <= n; i += pool_size) {
      x = h * ((double)i - 0.5);
      sum += f(x);
   }
   mypi = h * sum;

   MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, MASTER_RANK,
              MPI_COMM_WORLD);

   if (i_am_the_master) printf("\npi is approximately %.16f\n", pi);

   MPI_Finalize ();
}
```

Here is how this program compiles and runs:

```
gustav@sp19:../MPI 21:56:02 !520 $ mpcc -o pi pi.c
gustav@sp19:../MPI 21:56:59 !521 $ cat pi.ll
# @ shell = /afs/ovpit.indiana.edu/@sys/gnu/bin/bash
# @ job_type = parallel
# @ environment = COPY_ALL; MP_EUILIB=ip; MP_INFOLEVEL=2
# @ requirements = (Adapter == "hps_ip") && (Machine != "sp20") \
                   && (Machine != "sp18")
# @ min_processors = 4
# @ max_processors = 8
# @ output = pi.out
# @ error = pi.err
# @ class = test
# @ queue
poe pi << EOF
300
EOF
gustav@sp19:../MPI 21:57:08 !522 $ llsubmit pi.ll
submit: The job "sp19.107" has been submitted.
gustav@sp19:../MPI 21:57:22 !523 $ cat pi.out
Enter the number of intervals:
pi is approximately 3.1415935795157193
gustav@sp19:../MPI 21:57:59 !524 $
```

The synopsis of the program:

- This program evaluates π by numerically evaluating the integral

$$\int_0^1 \frac{1}{1+x^2}\,\mathrm{d}x = \frac{\pi}{4}$$

- The master process reads number of intervals from standard input, this number is then broadcast to the pool of processes.
- Having received the number of intervals, each process evaluates the total area of $n/$`pool_size` rectangles under the curve.
- The contributions to the total area under the curve are collected from participating processes by the master process, which at the same time adds them up, and prints the result on standard output.

Now let us discuss the program in more detail.

Assume, as before, that you are one of the processes. So first you find out about the number of processes in the pool and then about your own rank number. Then you check if you happen to have been annointed to be the Master, and if you are, then you communicate with The User:

```
if (i_am_the_master) {
   printf("Enter the number of intervals: ");
   scanf("%d",&n);
   if (n==0) n=100;
}
```

Having obtained from The User the total number of intervals, you broadcast that number to all other processes in the pool thusly:

```
MPI_Bcast(&n, 1, MPI_INT, MASTER_RANK, MPI_COMM_WORLD);
```

Now everybody gets down to work. Every process finds the width of an interval:

```
h   = 1.0 / (double) n;
```

Initialises its own `sum` to zero, and commences the following computation:

```
for (i = my_rank + 1; i <= n; i += pool_size) {
   x = h * ((double)i - 0.5);
   sum += f(x);
}
```

What happens here is as follows. First you find the value of x at the middle of an interval. Then evaluate $4/(1+x^2)$ and *add* it to whatever has already been accumulated in *your* `sum`. Then you jump over to another interval, which is `pool_size` intervals to the right and repeat the operation. In the meantime other processes will work on their own intervals, and when all is said and done, each of you will have a portion of the total sum in your local `sum`. In order to convert that portion of the total sum into a portion of the total integral, you need to multiply that local `sum` by the width of the interval, which is h:

```
  mypi = h * sum;
```

What you have at this stage is not the real π. It is the portion of π. All those portions have to be added together to get the real π. This is done by sending your contributions to the master, who performs the addition. The operation that accomplishes this is the reduction operation:

```
  MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, MASTER_RANK,
             MPI_COMM_WORLD);
```

In plain English the meaning of the above is as follows:

> *All processes in the* `MPI_COMM_WORLD`*, including the master process, send one item of type* `MPI_DOUBLE` *from their buffer called* `mypi` *to the process whose rank is* `MASTER_RANK`*, i.e., to the master process. The master process performs the* `MPI_SUM` *operation on those contributions, i.e., adds them all together, and writes the result on its own buffer called* `pi`*. The master process is the only one here, who knows the final outcome of the operation.*

And so, the master process is also the one that writes it on standard output.

MPI provides a number of predefined reduction operations that can be used in this context, and you can define your own operations too. The predefined ones are:

MPI_MAX	MPI_MIN	MPI_SUM
MPI_PROD	MPI_LAND	MPI_BAND
MPI_LOR	MPI_BOR	MPI_LXOR
MPI_BXOR	MPI_MAXLOC	MPI_MINLOC

8.2.4 Bank Queue

The following program illustrates a rather important parallel programming technique, which is often referred to as a *job queue* or a *bank queue* paradigm. The idea is as follows: you have a number of jobs that you need to attend to, and that are not dependent on each other. The master process maintains the job queue, and sends jobs to slave processes. The slaves labour on the jobs and return the results back to the master. Once a slave has finished working on its last assignment and returned the results to the master, a new job is sent to the slave. That way all processes of an MPI farm are kept busy.

The paradigm can be enriched. For example slave processes can return not only answers to the master, but also new jobs. The master process may assess those jobs, perhaps compare them to a list of jobs already done that it may keep on a lightweight data base, and if a job is new indeed, it can be added to the queue, whereas if there is already an answer available to that job, the answer may be passed to the slave together with the information that this class of problems has already been solved.

That way the queue can grow and shrink dynamically as the computation proceeds.

This is a very dynamic paradigm and it is often used to traverse dynamically growing trees, especially in artificial reasoning programs.

The master itself may take part in the work, other than just maintaining the queue and communicating with the slaves. But if there is a lot of work and a lot of slaves, the master process may be too busy to carry on with any computation of its own.

When you write programs like that you must also remember that communication is expensive. Consequently jobs sent to the slaves must be substantial enough to occupy them for a very long time. It is very easy to overload the master process if task granularity is too small. In that case the communication and the inability of the master process to respond quickly enough to slaves' requests can become a bottle neck.

Here's the program itself. In this program we are simply going to multiply matrix $\boldsymbol{A}$ by vector $\boldsymbol{b}$ in parallel. Every slave process will have its own copy of $\boldsymbol{b}$, and the master is going to send them rows of $\boldsymbol{A}$ to multiply by their own copy of $\boldsymbol{b}$. The result is a corresponding entry in vector $\boldsymbol{c} = \boldsymbol{A} \cdot \boldsymbol{b}$. A slave process are going to deliver the entry to the master process, which will then place it in an appropriate slot in $\boldsymbol{c}$ and pass a new row to the slave process at the same time, if there is any more work still to be done. If there is no more work, the master sacks the slave process, but sending it a termination message.

So here is the code:

```
gustav@sp20:../MPI 18:19:12 !522 $ cat bank.c
#include <stdio.h>
#include <mpi.h>
#define COLS 100
#define ROWS 100
#define TRUE 1
#define FALSE 0
#define MASTER_RANK 0

int main ( int argc, char **argv )
{
   int pool_size, my_rank, destination;
   int i_am_the_master = FALSE;
   int a[ROWS][COLS], b[ROWS], c[ROWS], i, j;
   int int_buffer[BUFSIZ];
   MPI_Status status;


   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &pool_size);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   if (my_rank == MASTER_RANK) i_am_the_master = TRUE;

   if (i_am_the_master) {

      int row, count, sender;
      for (i = 0; i < COLS; i++) {
         b[i] = 1;
         for (j = 0; j < ROWS; j++) a[i][j] = i;
      }

      MPI_Bcast(b, ROWS, MPI_INT, MASTER_RANK, MPI_COMM_WORLD);

      count = 0;
      for (destination = 0; destination < pool_size; destination++) {
         if (destination != my_rank) {
            for (j = 0; j < COLS; j++) int_buffer[j] = a[count][j];
            MPI_Send(int_buffer, COLS, MPI_INT, destination, count,
                     MPI_COMM_WORLD);
            printf("sent row %d to %d\n", count, destination);
            count = count + 1;
         }
      }

      for (i = 0; i < ROWS; i++) {
         MPI_Recv (int_buffer, BUFSIZ, MPI_INT, MPI_ANY_SOURCE,
                   MPI_ANY_TAG, MPI_COMM_WORLD, &status);
         sender = status.MPI_SOURCE;
         row = status.MPI_TAG;
         c[row] = int_buffer[0];
         printf("\treceived row %d from %d\n", row, sender);
         if (count < ROWS) {
            for (j = 0; j < COLS; j++) int_buffer[j] = a[count][j];
            MPI_Send(int_buffer, COLS, MPI_INT, sender, count,
                     MPI_COMM_WORLD);
            printf("sent row %d to %d\n", count, sender);
            count = count + 1;
         }
         else {
            MPI_Send(NULL, 0, MPI_INT, sender, ROWS, MPI_COMM_WORLD);
            printf("terminated process %d with tag %d\n", sender, ROWS);
         }
      }
   }
   else { /* I am not the master */

      int sum, row;
      FILE *log_file;

      log_file = fopen ("/tmp/gustav_log", "w");

      MPI_Bcast(b, COLS, MPI_INT, MASTER_RANK, MPI_COMM_WORLD);
      fprintf(log_file, "received broadcast from %d\n", MASTER_RANK);
      fflush(log_file);
      MPI_Recv(int_buffer, COLS, MPI_INT, MASTER_RANK, MPI_ANY_TAG,
                 MPI_COMM_WORLD, &status);
      fprintf(log_file, "received a message from %d, tag %d\n",
                status.MPI_SOURCE, status.MPI_TAG);
      fflush(log_file);
      while (status.MPI_TAG != ROWS) { /* The job is not finished */
         row = status.MPI_TAG; sum = 0;
         for (i = 0; i < COLS; i++) sum = sum + int_buffer[i] * b[i];
         int_buffer[0] = sum;
         MPI_Send (int_buffer, 1, MPI_INT, MASTER_RANK, row, MPI_COMM_WORLD);
         fprintf(log_file, "sent row %d to %d\n", row, MASTER_RANK);
         fflush(log_file);
         MPI_Recv (int_buffer, COLS, MPI_INT, MASTER_RANK, MPI_ANY_TAG,
                   MPI_COMM_WORLD, &status);
         fprintf(log_file, "received a message from %d, tag %d\n",
                 status.MPI_SOURCE, status.MPI_TAG);
         fflush(log_file);
      }
      fprintf(log_file, "exiting on  tag %d\n", status.MPI_TAG);
      fflush(log_file);
   }

   MPI_Finalize ();
}

gustav@sp20:../MPI 18:19:16 !523 $
```

And here is how this code compiles and runs:

```
gustav@sp20:../MPI 18:20:01 !524 $ mpcc -o bank bank.c
gustav@sp20:../MPI 18:20:10 !525 $ cat bank.ll
# @ job_type = parallel
# @ environment = COPY_ALL; MP_EUILIB=ip; MP_INFOLEVEL=2
# @ requirements = (Adapter == "hps_ip") && (Machine != "sp20") \
                   && (Machine != "sp18")
# @ min_processors = 4
# @ max_processors = 8
# @ class = test
# @ notification = never
# @ executable = /usr/bin/poe
# @ arguments = bank
# @ output = bank.out
# @ error = bank.err
# @ queue
gustav@sp20:../MPI 18:20:15 !526 $ llsubmit bank.ll
submit: The job "sp20.98" has been submitted.
gustav@sp20:../MPI 18:20:18 !527 $
```

The results are returned on `bank.out`, which looks as follows:

```
gustav@sp20:../MPI 18:21:10 !529 $ cat bank.out
sent row 0 to 1
sent row 1 to 2
sent row 2 to 3
sent row 3 to 4
sent row 4 to 5
sent row 5 to 6
sent row 6 to 7
        received row 1 from 2
sent row 7 to 2
        received row 2 from 3
sent row 8 to 3
        received row 7 from 2
sent row 9 to 2
        received row 8 from 3
```

```
sent row 10 to 3
        received row 9 from 2
sent row 11 to 2
        received row 10 from 3
sent row 12 to 3
        received row 11 from 2
sent row 13 to 2
        received row 12 from 3
sent row 14 to 3
        received row 13 from 2
sent row 15 to 2
        received row 14 from 3
sent row 16 to 3
        received row 15 from 2
sent row 17 to 2
        received row 16 from 3
sent row 18 to 3
        received row 17 from 2
sent row 19 to 2
        received row 18 from 3
sent row 20 to 3
        received row 19 from 2
sent row 21 to 2
        received row 20 from 3
sent row 22 to 3
        received row 21 from 2
sent row 23 to 2
        received row 22 from 3
sent row 24 to 3
        received row 23 from 2
sent row 25 to 2
        received row 24 from 3
sent row 26 to 3
        received row 25 from 2
sent row 27 to 2
        received row 26 from 3
sent row 28 to 3
        received row 27 from 2
sent row 29 to 2
        received row 28 from 3
sent row 30 to 3
        received row 29 from 2
sent row 31 to 2
        received row 30 from 3
sent row 32 to 3
        received row 31 from 2
sent row 33 to 2
        received row 32 from 3
sent row 34 to 3
        received row 33 from 2
sent row 35 to 2
        received row 34 from 3
sent row 36 to 3
        received row 35 from 2
sent row 37 to 2
        received row 36 from 3
sent row 38 to 3
```

```
        received row 3 from 4
sent row 39 to 4
        received row 4 from 5
sent row 40 to 5
        received row 37 from 2
sent row 41 to 2
        received row 38 from 3
sent row 42 to 3
        received row 5 from 6
sent row 43 to 6
        received row 6 from 7
sent row 44 to 7
        received row 39 from 4
sent row 45 to 4
        received row 40 from 5
sent row 46 to 5
        received row 41 from 2
sent row 47 to 2
        received row 42 from 3
sent row 48 to 3
        received row 43 from 6
sent row 49 to 6
        received row 44 from 7
sent row 50 to 7
        received row 45 from 4
sent row 51 to 4
        received row 46 from 5
sent row 52 to 5
        received row 47 from 2
sent row 53 to 2
        received row 48 from 3
sent row 54 to 3
        received row 49 from 6
sent row 55 to 6
        received row 50 from 7
sent row 56 to 7
        received row 51 from 4
sent row 57 to 4
        received row 52 from 5
sent row 58 to 5
        received row 53 from 2
sent row 59 to 2
        received row 54 from 3
sent row 60 to 3
        received row 55 from 6
sent row 61 to 6
        received row 56 from 7
sent row 62 to 7
        received row 57 from 4
sent row 63 to 4
        received row 58 from 5
sent row 64 to 5
        received row 59 from 2
sent row 65 to 2
        received row 60 from 3
sent row 66 to 3
        received row 61 from 6
```

```
sent row 67 to 6
        received row 62 from 7
sent row 68 to 7
        received row 63 from 4
sent row 69 to 4
        received row 64 from 5
sent row 70 to 5
        received row 65 from 2
sent row 71 to 2
        received row 66 from 3
sent row 72 to 3
        received row 67 from 6
sent row 73 to 6
        received row 68 from 7
sent row 74 to 7
        received row 69 from 4
sent row 75 to 4
        received row 70 from 5
sent row 76 to 5
        received row 71 from 2
sent row 77 to 2
        received row 72 from 3
sent row 78 to 3
        received row 73 from 6
sent row 79 to 6
        received row 75 from 4
sent row 80 to 4
        received row 74 from 7
sent row 81 to 7
        received row 76 from 5
sent row 82 to 5
        received row 77 from 2
sent row 83 to 2
        received row 78 from 3
sent row 84 to 3
        received row 79 from 6
sent row 85 to 6
        received row 80 from 4
sent row 86 to 4
        received row 81 from 7
sent row 87 to 7
        received row 82 from 5
sent row 88 to 5
        received row 83 from 2
sent row 89 to 2
        received row 84 from 3
sent row 90 to 3
        received row 85 from 6
sent row 91 to 6
        received row 86 from 4
sent row 92 to 4
        received row 87 from 7
sent row 93 to 7
        received row 88 from 5
sent row 94 to 5
        received row 89 from 2
sent row 95 to 2
```

```
        received row 90 from 3
sent row 96 to 3
        received row 91 from 6
sent row 97 to 6
        received row 92 from 4
sent row 98 to 4
        received row 93 from 7
sent row 99 to 7
        received row 94 from 5
terminated process 5 with tag 100
        received row 95 from 2
terminated process 2 with tag 100
        received row 96 from 3
terminated process 3 with tag 100
        received row 97 from 6
terminated process 6 with tag 100
        received row 98 from 4
terminated process 4 with tag 100
        received row 99 from 7
terminated process 7 with tag 100
        received row 0 from 1
terminated process 1 with tag 100
gustav@sp20:../MPI 18:21:16 !530 $
```

Let me now explain in detail how this program works.
The matrix is 100×100, which is fixed by

```
#define COLS 100
#define ROWS 100
```

and the master process is going to be the process of rank 0:

```
#define MASTER_RANK 0
```

After the initial incantations:

```
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &pool_size);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
```

all processes know about the size of the process pool and their own rank within it, including the master process, which asserts:

```
    if (my_rank == MASTER_RANK) i_am_the_master = TRUE;
```

Now we enter an interesting part of the code. There is a large `if` statement there, which looks as follows:

```
   if (i_am_the_master) {
      blah... blah... blah..
   }
   else { /* I am not the master */
      blah... blah... blah...
   }

   MPI_Finalize ();
```

As you see, this `if` statement divides the code into two subprograms: the master process executes the first clause of `if` and the slave processes execute the `else` clause.

The two subprograms are really quite different and they don't merge at all until the final `MPI_Finalize()`.

The Master Program

The first thing that the master process does is to initialize vector $\boldsymbol{b}$ and matrix $\boldsymbol{A}$. Vector $\boldsymbol{b}$ is set to 1 and matrix $\boldsymbol{A}$ is set to $A_{ij} = i$:

```
for (i = 0; i < COLS; i++) {
   b[i] = 1;
   for (j = 0; j < ROWS; j++) a[i][j] = i;
}
```

Then the master process broadcasts $\boldsymbol{b}$ to all processes (including itself):

```
MPI_Bcast(b, ROWS, MPI_INT, MASTER_RANK, MPI_COMM_WORLD);
```

Now the master process initializes the counter, `count`, which is going to be used to number rows sent to the slave processes, and send the first batch of jobs to *all* slave processes. In this program the master does not participate in the computation, so it does not send a job to itself:

```
count = 0;
for (destination = 0; destination < pool_size; destination++) {
   if (destination != my_rank) {
      for (j = 0; j < COLS; j++) int_buffer[j] = a[count][j];
      MPI_Send(int_buffer, COLS, MPI_INT, destination, count,
               MPI_COMM_WORLD);
      printf("sent row %d to %d\n", count, destination);
      count = count + 1;
   }
}
```

For clarity, I have made the master process transfer a row from $\boldsymbol{A}$ to a send buffer called `int_buffer`, and then send the data to the slave process. The data could be sent directly from matrix $\boldsymbol{A}$, which would be faster, but then you would have to remember how C stores matrices (it stores them in a row-major fashion, so this would actually work in this program).

Observe also that I have used `count` in place of the message *tag*. The slave processes, as you will see later, will use the tag number associated with the message in order to figure out, which particular row of matrix $\boldsymbol{A}$ they are about to work with. Now, the slaves don't really have to know that, but the master does. So when the slave sends the answer back to the master, it will use the same tag number, to remind the master process about the row number that the answer corresponds to.

Every time a matrix row is sent to a slave process, the master process logs it on standard output.

The following `for` loop is the tricky part of the master program. The master process waits for a message to arrive, from any process, and from any source:

how is the master process to know, which slave process is going to be the first with an answer. Some may be slower and busier than others depending on what else runs on their CPUs.

Once a message has arrived, the master process checks where the message has come from by inspecting the `status` structure associated with the message:

```
sender = status.MPI_SOURCE;
```

and inspects the tag number of the message, so that it is reminded about the row number that the answer relates to:

```
row = status.MPI_TAG;
```

The answer itself is then placed in an appropriate slot of vector $\boldsymbol{c}$:

```
c[row] = int_buffer[0];
```

and the whole operation logged on standard output:

```
printf("\treceived row %d from %d\n", row, sender);
```

Now the master process has to respond to the slave process that was so kind to deliver the answer. The master process checks if there is still any work left:

```
if (count < ROWS) {
   blah... blah... blah...
}
else {
   MPI_Send(NULL, 0, MPI_INT, sender, ROWS, MPI_COMM_WORLD);
   printf("terminated process %d with tag %d\n", sender, ROWS);
}
```

and if there isn't, it sends a null message to the slave process in question, whose tag is `ROWS`, and logs it on standard output.

Now, in C arrays are numbered from 0 through length $-$ 1, and all processes know that matrix $\boldsymbol{A}$ has `ROWS` rows, numbered from 0 through `ROWS`-1. So if the tag of the message is `ROWS` the slave process is going to know that something's amiss. As a matter of fact it will know that this is a termination message, so it will go away and terminate itself.

If there is still some work left though, then the master process transfers the corresponding row of matrix $\boldsymbol{A}$ to its send buffer, `int_buffer`, and sends its content to the slave process that has just delivered the answer. This operation is again logged on standard output, and the counter, that counts how many rows of matrix $\boldsymbol{A}$ have been sent out so far, is incremented by 1:

```
for (j = 0; j < COLS; j++) int_buffer[j] = a[count][j];
MPI_Send(int_buffer, COLS, MPI_INT, sender, count,
         MPI_COMM_WORLD);
printf("sent row %d to %d\n", count, sender);
count = count + 1;
```

After the master process has collected answers to all problems that it sent to the slave processes, there is nothing else left for it to do, so it hits `MPI_Finalize()` and terminates itself together with all the slaves that by now should have been waiting for her at the barrier.

They jump off the cliff together.

The Slave Program

Let use have a look at the slave program now that is enclosed in

```
else { /* I am not the master */
   blah... blah... blah...
}
```

The slave processes begin their career by opening a log file on their own *local* `/tmp` directory. In this program the file is called simply `gustav_log`, because I couldn't think of anything else - but in a serious application you may have to put up an effort and generate a unique file name with some generic prefix, perhaps.

In this program all processes write a lot, so that you can see what they've done. If they were to write it all on the same standard output as the master process, the whole information would get garbled and quite useless.

The next step is to *receive* vector $\boldsymbol{b}$, which has been sent by the master.

Why do we call `MPI_Bcast` in this program twice? Well, we don't. It only appears so, because `MPI_Bcast` is printed twice within the text of the program. But the call to `MPI_Bcast` issued within the master part would not have been executed by the slaves, so here we have to type the call again, separately, for the slave processes.

After they have received their copies of $\boldsymbol{b}$, they log this event on their respective log files and wait for the first batch of jobs to be sent to them by the master process.

Having received their first row of $\boldsymbol{A}$ they log it on `gustav_log` and commence work.

The work is done within the large

```
      while (status.MPI_TAG != ROWS) { /* The job is not finished */
         blah... blah... blah...
      }
```

loop. Every time a slave process receives a message from the master process it checks if the tag of the message is less than `ROWS`. Remember that having received a message with tag `ROWS` implies the termination of the contract!

If the tag is kosher, the slave process does the following:

```
         row = status.MPI_TAG; sum = 0;
         for (i = 0; i < COLS; i++) sum = sum + int_buffer[i] * b[i];
         int_buffer[0] = sum;
         MPI_Send (int_buffer, 1, MPI_INT, MASTER_RANK, row, MPI_COMM_WORLD);
         fprintf(log_file, "sent row %d to %d\n", row, MASTER_RANK);
         fflush(log_file);
```

The row number is extracted from the tag of the message. Then the slave process evaluates $\sum_j A_{ij} b_j$ and sends it back to the master using the same tag. So that the master will know which row number the answer corresponds to.

This operation, again, is logged on `/tmp/gustav_log`.

Finally, the slave process waits for another message from the master process, reads it, and logs this operation on `gustav_log`:

```
MPI_Recv (int_buffer, COLS, MPI_INT, MASTER_RANK, MPI_ANY_TAG,
          MPI_COMM_WORLD, &status);
fprintf(log_file, "received a message from %d, tag %d\n",
        status.MPI_SOURCE, status.MPI_TAG);
fflush(log_file);
```

Then it's back to the top of the loop: check the tag, if the tag is OK perform the computation, otherwise hit `MPI_Finalize()`.

It is instructive to have a look at one of those files generated by slave processes.

When I have run this program on our SP, I got the following in my `bank.err` file:

```
INFO: 0031-119  Host sp40.ucs.indiana.edu allocated for task 0
INFO: 0031-119  Host sp22.ucs.indiana.edu allocated for task 1
INFO: 0031-119  Host sp19.ucs.indiana.edu allocated for task 2
INFO: 0031-119  Host sp17.ucs.indiana.edu allocated for task 3
INFO: 0031-119  Host sp42.ucs.indiana.edu allocated for task 4
INFO: 0031-119  Host sp43.ucs.indiana.edu allocated for task 5
INFO: 0031-119  Host sp23.ucs.indiana.edu allocated for task 6
INFO: 0031-119  Host sp41.ucs.indiana.edu allocated for task 7
```

This tells me that my master ran on node `sp40`, and the slaves ran on nodes `sp17`, `sp19`, `sp22`, `sp23`, `sp41`, `sp42`, and `sp43`.

So let's go to, say, `sp41`, and have a look at what's in `/tmp`:

```
gustav@sp41:../SP 19:36:25 !501 $ cd /tmp
gustav@sp41:..//tmp 19:36:26 !502 $ ls
gustav_log                 ssh-gustav                 startd_unix_dgram_socket
gustav@sp41:..//tmp 19:36:28 !503 $ cat gustav_log
received broadcast from 0
received a message from 0, tag 6
sent row 6 to 0
received a message from 0, tag 44
sent row 44 to 0
received a message from 0, tag 50
sent row 50 to 0
received a message from 0, tag 56
sent row 56 to 0
received a message from 0, tag 62
sent row 62 to 0
received a message from 0, tag 68
sent row 68 to 0
received a message from 0, tag 74
sent row 74 to 0
received a message from 0, tag 81
sent row 81 to 0
```

```
received a message from 0, tag 87
sent row 87 to 0
received a message from 0, tag 93
sent row 93 to 0
received a message from 0, tag 99
sent row 99 to 0
received a message from 0, tag 100
exiting on  tag 100
gustav@sp41:..//tmp 19:36:35 !504 $
```

What we find from this log is that the process running on node `sp41` received row number 6 initially, and took a rather long time to return the answer to the master process, and by the time it did that, the other processes have nearly finished half of the matrix. But from that point onwards `sp41` worked quite conscientiously on roughly every 6th row.

The beauty of the *job queue paradigm* is that you keep all processes as busy as they can get, even if some have to cope with more load than others.

8.3 Not So Simple MPI

The four programs we had a look at so far had been all implemented using a minimal set of MPI calls, plus broadcast and reduce.

In this section we are going to look at two more involved examples that will illustrate the following:

- Process topology
    - Cartesian communicators
- MPI data types and their definitions
- Gather and scatter operations

The first example will also illustrate how to do things in MPI that are done for you automatically by HPF. This example provides a very good comparison, and it illustrates very succinctly the power of HPF, for those problems, of course, that are tractable using data parallelism. If your problems do not fit in this category, then you may have little choice but to grit your teeth and grapple with MPI.

8.3.1 The Diffusion Problem

In this section I will lay the ground for an MPI version of a diffusion code based on Jacobi iterations. The idea is that a flat rectangular region, whose dimensions in this token example are going to be 6×8, will be divided amongst 12 processes, each handling a small square 2×2. But because it is a differential problem, apart from the 2×2 data squares, the processes will also have to maintain additional data that is going to mirror data that corresponds to whatever the neighbouring processes have in their 2×2 squares.

In this example we assume that we are interested only in one layer of data from the neighbours. Because every square, with the exception of the ones at the boundary of the region, is going to have four neighbours, we will have to add either a row or a column to our 2×2 squares in order to accomodate data transferred from neighbouring processes. Consequently each process will have to look after a 4×4 integer matrix, of which an internal 2×2 matrix represents that process' own data, and the boundaries of the matrix represent data obtained from the neighbours.

All that we are going to do within this program is to

1. arrange processes into a 2-dimensional Cartesian process topology
2. orient processes within this new topology
3. exchange data with neighbouring processes within the Cartesian topology

At every stage the master process is going to collect data from all other processes and print it on standard output so as to show the state of the system at one glance.

In order to help you understand the whole procedure, I'm going to show you some of the output of the program first, then I'll list the program for you, and then discuss it in more detail.

When the program starts, right after the processes have formed the new Cartesian topology, oriented themselves within it, but before they exchanged any data, their state looks as follows:

```
 9  9  9  9   10 10 10 10   11 11 11 11
 9  9  9  9   10 10 10 10   11 11 11 11
 9  9  9  9   10 10 10 10   11 11 11 11
 9  9  9  9   10 10 10 10   11 11 11 11

 6  6  6  6    7  7  7  7    8  8  8  8
 6  6  6  6    7  7  7  7    8  8  8  8
 6  6  6  6    7  7  7  7    8  8  8  8
 6  6  6  6    7  7  7  7    8  8  8  8

 3  3  3  3    4  4  4  4    5  5  5  5
 3  3  3  3    4  4  4  4    5  5  5  5
 3  3  3  3    4  4  4  4    5  5  5  5
 3  3  3  3    4  4  4  4    5  5  5  5

 0  0  0  0    1  1  1  1    2  2  2  2
 0  0  0  0    1  1  1  1    2  2  2  2
 0  0  0  0    1  1  1  1    2  2  2  2
 0  0  0  0    1  1  1  1    2  2  2  2
```

Every process initializes its own 4×4 matrix to its own rank number. The matrices are displayed by the master process in a way that illustrates the topology of the whole system, i.e., we have a rectangular topology 3×4, process rank 0 is in the lower left corner, and its neighbours are process rank 1 on the right and process rank 3 above. Process rank 4 sits roughly in the middle and its neighbours are process rank 1 below, process rank 7 above, process rank 3 on the left, and process rank 5 on the right.

Now, the processes exchange the content of their *inner* rows with their neighbours, i.e., process rank 4 sends the content of its row 3 (counted from the bottom) to process number 7, which puts it in its own row 1 (counted from the bottom), and, at the same time receives the content of row 2 from process rank 7, and places it in its own row 4.

So that after this exchange operation has taken place, the matrices look as follows:

```
 9  9  9  9   10 10 10 10   11 11 11 11
 9  9  9  9   10 10 10 10   11 11 11 11
 9  9  9  9   10 10 10 10   11 11 11 11
 6  6  6  6    7  7  7  7    8  8  8  8

 9  9  9  9   10 10 10 10   11 11 11 11
 6  6  6  6    7  7  7  7    8  8  8  8
 6  6  6  6    7  7  7  7    8  8  8  8
 3  3  3  3    4  4  4  4    5  5  5  5

 6  6  6  6    7  7  7  7    8  8  8  8
 3  3  3  3    4  4  4  4    5  5  5  5
 3  3  3  3    4  4  4  4    5  5  5  5
 0  0  0  0    1  1  1  1    2  2  2  2

 3  3  3  3    4  4  4  4    5  5  5  5
 0  0  0  0    1  1  1  1    2  2  2  2
 0  0  0  0    1  1  1  1    2  2  2  2
 0  0  0  0    1  1  1  1    2  2  2  2
```

Now we have to repeat this operation, but this time exchanging columns between neighbours. And so process rank 4 will send its column 2 (counted from the left) to process rank 3, which is going to place it in its own column 4 (counted from the left), and, at the same time process rank 3 is going to send its own column 3 to process rank 4, which is going to place it in its own column 1.

And after all this is over, the matrices are going to look as follows:

```
 9  9  9 10    9 10 10 11   10 11 11 11
 9  9  9 10    9 10 10 11   10 11 11 11
 9  9  9 10    9 10 10 11   10 11 11 11
 6  6  6  7    6  7  7  8    7  8  8  8

 9  9  9 10    9 10 10 11   10 11 11 11
 6  6  6  7    6  7  7  8    7  8  8  8
 6  6  6  7    6  7  7  8    7  8  8  8
 3  3  3  4    3  4  4  5    4  5  5  5

 6  6  6  7    6  7  7  8    7  8  8  8
 3  3  3  4    3  4  4  5    4  5  5  5
 3  3  3  4    3  4  4  5    4  5  5  5
 0  0  0  1    0  1  1  2    1  2  2  2

 3  3  3  4    3  4  4  5    4  5  5  5
```

```
  0  0  0  1   0  1  1  2   1  2  2  2
  0  0  0  1   0  1  1  2   1  2  2  2
  0  0  0  1   0  1  1  2   1  2  2  2
```

Observe that now not only does every process know what data is harboured by its neighbours on the left, on the right, above, and below, but they even know the data that somehow made it diagonally, i.e., from the upper left, upper right, lower left, and lower right directions – but the latter is not going to last, because that data came from the mirror regions, so it is not truly representative of the processes' internal state.

Now every process can perform its own Jacobi iteration within its own little patch for a diffusion problem, and set new values to its own data, while keeping the mirrored data unchanged.

Before the next iteration can commence, the data has to be exchanged between the neighbours again, in order to refresh the mirrors.

So now let us have a look at the code:

The Code

```
#include <stdio.h>
#include <mpi.h>

#define FALSE 0
#define TRUE  1
#define MASTER_RANK 0

#define UPDOWN              0
#define SIDEWAYS            1
#define RIGHT               1
#define UP                  1

#define PROCESS_DIMENSIONS  2

#define PROCESS_ROWS        4
#define ROWS                4
#define DISPLAY_ROWS       16      /* must be PROCESS_ROWS * ROWS */

#define PROCESS_COLUMNS     3
#define COLUMNS             4
#define DISPLAY_COLUMNS    12      /* must be PROCESS_COLUMNS * COLUMNS */

int main ( int argc, char **argv )
{
   int pool_size, my_rank, destination, source;
   MPI_Status status;
   char char_buffer[BUFSIZ];
   int i_am_the_master = FALSE;

   int divisions[PROCESS_DIMENSIONS] = {PROCESS_ROWS, PROCESS_COLUMNS};
   int periods[PROCESS_DIMENSIONS] = {0, 0};
   int reorder = 1;
   MPI_Comm cartesian_communicator;
   int my_cartesian_rank, my_coordinates[PROCESS_DIMENSIONS];
   int left_neighbour, right_neighbour, bottom_neighbour, top_neighbour;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &pool_size);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   if (my_rank == MASTER_RANK) i_am_the_master = TRUE;

   MPI_Cart_create ( MPI_COMM_WORLD, PROCESS_DIMENSIONS, divisions,
                     periods, reorder, &cartesian_communicator );

   if (cartesian_communicator != MPI_COMM_NULL) {

      int matrix [ROWS][COLUMNS];
      int i, j;
      MPI_Datatype column_type;

      MPI_Comm_rank ( cartesian_communicator, &my_cartesian_rank );
      MPI_Cart_coords ( cartesian_communicator, my_cartesian_rank,
                        PROCESS_DIMENSIONS, my_coordinates );
      MPI_Cart_shift ( cartesian_communicator, SIDEWAYS, RIGHT,
                       &left_neighbour, &right_neighbour );
      MPI_Cart_shift ( cartesian_communicator, UPDOWN, UP,
                       &bottom_neighbour, &top_neighbour );

      if (! i_am_the_master ) {
         sprintf(char_buffer, "process %2d, cartesian %2d, \
coords (%2d,%2d), left %2d, right %2d, top %2d, bottom %2d",
                 my_rank, my_cartesian_rank, my_coordinates[0],
                 my_coordinates[1], left_neighbour, right_neighbour,
                 top_neighbour, bottom_neighbour);
         MPI_Send(char_buffer, strlen(char_buffer) + 1, MPI_CHAR,
                  MASTER_RANK, 3003, MPI_COMM_WORLD);
      }
      else {

         int number_of_c_procs, count;

         number_of_c_procs = divisions[0] * divisions[1];
         for (count = 0; count < number_of_c_procs - 1; count++) {
            MPI_Recv(char_buffer, BUFSIZ, MPI_CHAR, MPI_ANY_SOURCE, 3003,
                     MPI_COMM_WORLD, &status);
            printf ("%s\n", char_buffer);
         }
         printf( "process %2d, cartesian %2d, \
coords (%2d,%2d), left %2d, right %2d, top %2d, bottom %2d\n",
                 my_rank, my_cartesian_rank, my_coordinates[0],
                 my_coordinates[1], left_neighbour, right_neighbour,
                 top_neighbour, bottom_neighbour);
      }

      for ( i = 0; i < ROWS; i++ ) {
         for ( j = 0; j < COLUMNS; j++ ) {
            matrix [i][j] = my_cartesian_rank;
         }
      }
      if (my_cartesian_rank != MASTER_RANK )
         MPI_Send ( matrix, COLUMNS * ROWS, MPI_INT, MASTER_RANK, 3003,
                    cartesian_communicator );
      else
         collect_matrices ( cartesian_communicator, my_cartesian_rank,
                            matrix, 3003 );

      MPI_Sendrecv ( &matrix[ROWS - 2][0], COLUMNS, MPI_INT,
                     top_neighbour, 4004,
                     &matrix[0][0], COLUMNS, MPI_INT, bottom_neighbour,
                     4004,
                     cartesian_communicator, &status );

      MPI_Sendrecv ( &matrix[1][0], COLUMNS, MPI_INT, bottom_neighbour,
                     5005,
                     &matrix[ROWS - 1][0], COLUMNS, MPI_INT,
                     top_neighbour, 5005,
                     cartesian_communicator, &status );

      if (my_cartesian_rank != MASTER_RANK )
         MPI_Send ( matrix, COLUMNS * ROWS, MPI_INT, MASTER_RANK, 6006,
                    cartesian_communicator );
      else
         collect_matrices ( cartesian_communicator, my_cartesian_rank,
                            matrix, 6006 );

      MPI_Type_vector (ROWS, 1, COLUMNS, MPI_INT, &column_type);
      MPI_Type_commit (&column_type);

      MPI_Sendrecv ( &matrix[0][1], 1, column_type, left_neighbour, 7007,
                     &matrix[0][COLUMNS - 1], 1, column_type,
                     right_neighbour, 7007,
                     cartesian_communicator, &status );

      MPI_Sendrecv ( &matrix[0][COLUMNS - 2], 1, column_type,
                     right_neighbour, 8008,
                     &matrix[0][0], 1, column_type, left_neighbour, 8008,
                     cartesian_communicator, &status );

      if (my_cartesian_rank != MASTER_RANK )
         MPI_Send ( matrix, COLUMNS * ROWS, MPI_INT, MASTER_RANK, 9009,
                    cartesian_communicator );
      else
         collect_matrices ( cartesian_communicator, my_cartesian_rank,
                            matrix, 9009 );
   }

   MPI_Finalize ();
}

int print_array (int array [DISPLAY_ROWS] [DISPLAY_COLUMNS],
                          int vertical_break,
                          int horizontal_break)
{
   int k, l;

   printf ("\n");
   for (k = DISPLAY_ROWS - 1; k >= 0; k -- ) {
      for (l = 0; l < DISPLAY_COLUMNS; l ++ ) {
         if (l % horizontal_break == 0) printf (" ");
         printf ( "%2d ", array [k][l] );
      }
      printf ( "\n" );
      if (k % vertical_break == 0) printf ( "\n" );
   }
}

int collect_matrices (MPI_Comm cartesian_communicator,
                      int my_cartesian_rank,
                      int matrix[ROWS][COLUMNS],
                      int tag)
{
   int coordinates[PROCESS_DIMENSIONS];
   int client_matrix[ROWS][COLUMNS];
   int display[DISPLAY_ROWS][DISPLAY_COLUMNS];
   int i, j, k, l, source;
   MPI_Status status;

   for ( i = PROCESS_ROWS - 1; i >= 0; i -- ) {
      for ( j = 0; j < PROCESS_COLUMNS; j ++ ) {
         coordinates[0] = i;
         coordinates[1] = j;
         MPI_Cart_rank ( cartesian_communicator, coordinates,
                         &source );
         if (source != my_cartesian_rank) {
            MPI_Recv ( client_matrix, BUFSIZ, MPI_INT, source, tag,
                       cartesian_communicator, &status );
            for ( k = ROWS - 1; k >= 0; k -- ) {
               for ( l = 0; l < COLUMNS; l++ ) {
                  display [i * ROWS + k] [j * COLUMNS + l] =
                     client_matrix[k][l];
               }
            }
         }
         else {
            for ( k = ROWS - 1; k >= 0; k -- ) {
               for ( l = 0; l < COLUMNS; l ++ ) {
                  display [i * ROWS + k] [j * COLUMNS + l]
                     = matrix[k][l];
               }
            }
         }
      }
   }

   print_array (display, ROWS, COLUMNS);
}
```

The compilation and run of this code are done as follows:

```
gustav@sp20:../MPI 17:21:07 !527 $ mpcc -o cartesian cartesian.c
gustav@sp20:../MPI 17:21:08 !528 $ cat cartesian.ll
# @ job_type = parallel
# @ environment = COPY_ALL; MP_EUILIB=ip; MP_INFOLEVEL=2
# @ requirements = (Adapter == "hps_ip")
# @ min_processors = 12
# @ max_processors = 12
# @ executable = /usr/bin/poe
# @ arguments = cartesian
# @ output = cartesian.out
# @ error = cartesian.err
# @ class = test
# @ queue
gustav@sp20:../MPI 17:21:11 !529 $ llsubmit cartesian.ll
submit: The job "sp20.117" has been submitted.
gustav@sp20:../MPI 17:21:16 !530 $
```

If you can't get 12 nodes, ask for 9 nodes only and change the defines in the code as follows:

```
#define PROCESS_ROWS        3
#define ROWS                4
#define DISPLAY_ROWS       12      /* must be PROCESS_ROWS * ROWS */

#define PROCESS_COLUMNS     3
#define COLUMNS             4
#define DISPLAY_COLUMNS    12      /* must be PROCESS_COLUMNS * COLUMNS */
```

The Discussion

The code begins innocently enough like all other MPI codes that we have seen so far: MPI itself is initialised, then every process finds out about the size of the process pool, and its own rank within that pool.

```
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &pool_size);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
```

And then, as usual, one of the processes assumes the role of the master:

```
   if (my_rank == MASTER_RANK) i_am_the_master = TRUE;
```

But the next operation is entirely new:

```
   MPI_Cart_create ( MPI_COMM_WORLD, PROCESS_DIMENSIONS, divisions,
                     periods, reorder, &cartesian_communicator );
```

What happens here is as follows. The processes create a new communicator, which, unlike the `MPI_COMM_WORLD`, has a special topology imposed on it. The new communicator, the *place* for which is passed to `MPI_Cart_create` as the last argument, is formed out of the old one, which is passed to `MPI_Cart_create` in the first argument. The four parameters passed in the middle: `PROCESS_DIMENSIONS`, `divisions`, `periods`, and `reorder`, specify the topology of the new communicator. When we say *topology* it means that there is some sort of a connectivity between the processes within this new communicator. In this case they will acquire left, right, bottom, and top *neighbours*.

The value of PROCESS_DIMENSIONS is 2, which means that the processes are to be organised into a 2 dimensional grid. divisions is going to be an array of rank 1, whose entries specify lengths of the process grid in those two dimensions, in this case:

```
divisions[PROCESS_DIMENSIONS] = {4, 3};
```

that is, the processes are going to be organised into a rectangular grid with 4 rows and 3 columns.

The parameter periods specifies if the topology is going to be a wrap-around topology or an open ended topology, i.e., if the processes that are on the borders of the region should have as their *over the border* neighbours the guys on the other side, or nobody. In this case we simply say that

```
periods[PROCESS_DIMENSIONS] = {0, 0};
```

which means that we don't want any wrapping.

The last parameter in this group, reorder, specifies if processes should be re-ordered for efficiency. This implies that processes may be moved around between processors, or just renumbered, so as to utilize better any hardware architecture that a given machine may have. Examples of such machines are Cray T3E, Fujitsu AP-3000, and NEC Cenju-3.

If you have asked for a 4×3 Cartesian communicator and started with, say, 15 processes, then 3 of those 15 processes would have to be rejected from the new communicator, because there is only going to be enough space in it for 12 processes. The rejected processes will find that after MPI_Cart_create returns, the returned value of cartesian_communicator is MPI_COMM_NULL. What this means is that those processes didn't get the job.

They may hang about, if they choose, or they may go right for MPI_Finalize – this is up to the programmer. You may create more than one communicator with various properties, so that the processes that didn't make it into the cartesian_communicator will get jobs elsewhere.

For this reason the remainder of the program is one large if statement:

```
   if (cartesian_communicator != MPI_COMM_NULL) {
      blah... blah... blah...
   }

   MPI_Finalize ();
}
```

Now, assuming that you did get a job, as a process, with the cartesian_communicator, the first thing that you do is to find your new rank number within that new communicator:

```
      MPI_Comm_rank ( cartesian_communicator, &my_cartesian_rank );
```

And the reason for this is that your rank number in the cartesian_communicator is an entirely different thing from your old number in the MPI_COMM_WORLD. They're like a tax file number in Australia and a tax file number in the US.

But now, since the cartesian_communicator is a communicator with a topology, you can make inquiries about your neighbourhood:

```
      MPI_Cart_coords ( cartesian_communicator, my_cartesian_rank,
                        PROCESS_DIMENSIONS, my_coordinates );
      MPI_Cart_shift ( cartesian_communicator, SIDEWAYS, RIGHT,
                       &left_neighbour, &right_neighbour );
      MPI_Cart_shift ( cartesian_communicator, UPDOWN, UP,
                       &bottom_neighbour, &top_neighbour );
```

The first call to `MPI_Cart_coords` returns your Cartesian coordinates in an array `my_coordinates` of length `PROCESS_DIMENSIONS`. Naturally you have to tell `MPI_Cart_coords` who you are, in order to obtain that information. You have to pass on your new Cartesian rank number in the second slot of the function.

The following two calls to `MPI_Cart_shift` tell you about the rank numbers of your left and right neighbours and of your bottom and top neighbours. So the function `MPI_Cart_shift` doesn't really *shift* anything. It's more like looking up who your neighbours are. But the designers of MPI thought of it in terms of *shifting* rank numbers of your neighbours from the left and from the right and from above and from below into those containers, which you call `left_neighbour`, `right_neighbour`, `bottom_neighbour`, and `top_neighbour`.

This stuff is worth displaying, so what we do now is to pack all that information into a message and send it to the master process.

Now we can commence the communication that occurs within the `MPI_COMM_WORLD`:

```
      if (! i_am_the_master ) {
         sprintf(char_buffer, "process %2d, cartesian %2d, \
coords (%2d,%2d), left %2d, right %2d, top %2d, bottom %2d",
                 my_rank, my_cartesian_rank, my_coordinates[0],
                 my_coordinates[1], left_neighbour, right_neighbour,
                 top_neighbour, bottom_neighbour);
         MPI_Send(char_buffer, strlen(char_buffer) + 1, MPI_CHAR,
                  MASTER_RANK, 3003, MPI_COMM_WORLD);
      }
      else {

         int number_of_c_procs, count;

         number_of_c_procs = divisions[0] * divisions[1];
         for (count = 0; count < number_of_c_procs - 1; count++) {
            MPI_Recv(char_buffer, BUFSIZ, MPI_CHAR, MPI_ANY_SOURCE, 3003,
                     MPI_COMM_WORLD, &status);
            printf ("%s\n", char_buffer);
         }
         printf( "process %2d, cartesian %2d, \
coords (%2d,%2d), left %2d, right %2d, top %2d, bottom %2d\n",
                 my_rank, my_cartesian_rank, my_coordinates[0],
                 my_coordinates[1], left_neighbour, right_neighbour,
                 top_neighbour, bottom_neighbour);
      }
```

By now you should be able to figure out this part of the code for yourself, because we've done both `MPI_Send` and `MPI_Recv`.

Observe a subtle bug here: we are assuming that the master process is still going to hang around! In other words that the master process was included into the `cartesian_communicator`.

What will happen if it is not included?

In our case we are going to run this job requesting exactly the right number of processes, so the master process is guaranteed to be included into the `cartesian_communicator`. Having said that you may consider modifying the logic of the program so as to get rid of the bug.

Here is what the output looks like, once the master process gets down to it:

```
process  8, cartesian  8, coords ( 2, 2), left  7, right -3, top 11, bottom  5
process  4, cartesian  4, coords ( 1, 1), left  3, right  5, top  7, bottom  1
process  1, cartesian  1, coords ( 0, 1), left  0, right  2, top  4, bottom -3
process  5, cartesian  5, coords ( 1, 2), left  4, right -3, top  8, bottom  2
process 11, cartesian 11, coords ( 3, 2), left 10, right -3, top -3, bottom  8
process  9, cartesian  9, coords ( 3, 0), left -3, right 10, top -3, bottom  6
process  3, cartesian  3, coords ( 1, 0), left -3, right  4, top  6, bottom  0
process  7, cartesian  7, coords ( 2, 1), left  6, right  8, top 10, bottom  4
process  2, cartesian  2, coords ( 0, 2), left  1, right -3, top  5, bottom -3
process  6, cartesian  6, coords ( 2, 0), left -3, right  7, top  9, bottom  3
process 10, cartesian 10, coords ( 3, 1), left  9, right 11, top -3, bottom  7
process  0, cartesian  0, coords ( 0, 0), left -3, right  1, top  3, bottom -3
```

You should compare this with the matrix lay-out that I have printed above and you'll see that it all makes some sense.

Observe that cartesian rank numbers here are the same as the world rank numbers. But you must not count on it. It just happens to be so here, on the SP, for this particular program, and under this particular version of PSSP, MPI, etc. This is not an MPI requirement. Just the opposite, in fact.

What happens now is this simple piece of code:

```
   for ( i = 0; i < ROWS; i++ ) {
      for ( j = 0; j < COLUMNS; j++ ) {
         matrix [i][j] = my_cartesian_rank;
      }
   }
```

Every process simply initializes its own little matrix with its own cartesian rank number. The whole matrix is initialized, including the parts that are going to be dedicated to mirroring, along the borders of the matrix.

Once the matrices have been initialized, the processes send them to the master process, who dilligently collects them and displays on standard output:

```
   if (my_cartesian_rank != MASTER_RANK )
      MPI_Send ( matrix, COLUMNS * ROWS, MPI_INT, MASTER_RANK, 3003,
                 cartesian_communicator );
   else
      collect_matrices ( cartesian_communicator, my_cartesian_rank,
                         matrix, 3003 );
```

But observe that this time the communication takes place entirely within the `cartesian_communicator`, and the process that collects all matrices does not have to be the same process as the master process in the `MPI_COMM_WORLD` communicator.

Now the processes in the `cartesian_communicator` have to exchange information with their neighbours:

```
MPI_Sendrecv ( &matrix[ROWS - 2][0], COLUMNS, MPI_INT,
               top_neighbour, 4004,
               &matrix[0][0], COLUMNS, MPI_INT, bottom_neighbour,
               4004,
               cartesian_communicator, &status );

MPI_Sendrecv ( &matrix[1][0], COLUMNS, MPI_INT, bottom_neighbour,
               5005,
               &matrix[ROWS - 1][0], COLUMNS, MPI_INT,
               top_neighbour, 5005,
               cartesian_communicator, &status );
```

While sending this stuff up and down, we're making use of the fact that matrices in C are stored in the *row-major* fashion, i.e., the other way to Fortran. The first statement sends the whole second row from the top (I have numbered these matrices upside down, so as to stick to the usual $x \times y$ convention) to the top neighbour. The send buffer begins at `&matrix[ROWS - 2][0]` and is `COLUMNS` items long. The items are of type `MPI_INT` and the corresponding message is going to have tag 4004.

At the same time, every process *receives* into its bottom row data sent by its bottom neighbour. The beginning of the receive buffer is `&matrix[0][0]`, the buffer contains items of type `MPI_INT`, and the number of those items is `COLUMNS`.

Of course this communication must take place within the `cartesian_communicator`, because it is only within this communicator that the notion of a bottom and top neighbour makes sense.

The second `MPI_Sendrecv` operation does the same, but in the opposite direction, i.e., the data is now sent down, whereas previously it went up.

Having done that we display the state of the whole system again:

```
if (my_cartesian_rank != MASTER_RANK)
   MPI_Send ( matrix, COLUMNS * ROWS, MPI_INT, MASTER_RANK, 6006,
              cartesian_communicator );
else
   collect_matrices ( cartesian_communicator, my_cartesian_rank,
                      matrix, 6006 );
```

The last part of the code is a little tricky. Here we have to exchange columns between neighbours, but columns are not stored contiguous in C.

MPI provides a very powerful apparatus for situations like that.

First we define a stridden data type:

```
MPI_Type_vector (ROWS, 1, COLUMNS, MPI_INT, &column_type);
MPI_Type_commit (&column_type);
```

This data type is constructed as follows: it comprises `ROWS` items of type `MPI_INT`. The items are collected by picking up 1 item every `COLUMNS` steps from a contiguous storage. The description of this peculiar data type is placed into `column_type`. Once the type has been defined it must be *committed*. Here we give a parallel machine an opportunity to adjust its hardware or software logic in order to prepare for manipulating this new data type.

Now we can exchange columns between neighbours in a much the same way we did rows before:

```
MPI_Sendrecv ( &matrix[0][1], 1, column_type, left_neighbour, 7007,
               &matrix[0][COLUMNS - 1], 1, column_type,
               right_neighbour, 7007,
               cartesian_communicator, &status );

MPI_Sendrecv ( &matrix[0][COLUMNS - 2], 1, column_type,
               right_neighbour, 8008,
               &matrix[0][0], 1, column_type, left_neighbour, 8008,
               cartesian_communicator, &status );
```

The first operation sends *one* item of `column_type`, stored at `&matrix[0][1]` to the `left_neighbour`. The corresponding message has tag 7007. At the same time another single item of `column_type` is received from the `right_neighbour` and placed in `&matrix[0][COLUMNS - 1]`. Again, observe that we are not sending the border columns: we are sending internal columns, and we're receiving them into the border columns.

The second `MPI_Sendrecv` call does the same in the opposite direction, i.e., from left to right.

After the data has been exchanged, the state of the system is displayed again by calling:

```
if (my_cartesian_rank != MASTER_RANK)
   MPI_Send ( matrix, COLUMNS * ROWS, MPI_INT, MASTER_RANK, 9009,
              cartesian_communicator );
else
   collect_matrices ( cartesian_communicator, my_cartesian_rank,
                      matrix, 9009 );
```

So, this is what the program does.

It is messy and clumsy compared to an HPF program, where you don't have to bother about any of that at all. IBM HPF compiler, as a matter of fact, compiles your HPF program to an MPI program that does very much what this simple example code illustrates. To use HPF in place of MPI, wherever applicable, will save you a lot of hard work.

8.3.2 Interacting Particles

The code in this section illustrates not only certain new MPI elements, but also an interesting and quite fast algorithm for evaluating interactions between many particles. Unfortunately the algorithm which relies on the `MPI_Allgather` operation is costly in terms of communications. You need a very fast communication fabric for this to work.

So, how does it work?

Every process of the MPI job maintains an array with data pertaining to all particles, but it calculates evolution for only a group of particles that it is responsible for. Once the computation completes, the new updated states for all particles are exchanged amongst all participating processes. This is done by the `MPI_Allgather` operation. A lot of data is being shoved around when that happens, but, as the result all processes end up receiving updated invormation about all particles, so that they can commence the next iteration.

The Code

```
#include <stdio.h>
#include <math.h>
#include <mpi.h>

#define FALSE           0
#define TRUE            1
#define MASTER_RANK     0

#define MAX_PARTICLES 1000
#define MAX_PROCS      128
#define EPSILON         1.0E-10


int main ( int argc, char **argv )
{
   int pool_size, my_rank;
   int i_am_the_master = FALSE;
   extern double drand48();
   extern void srand48();

   typedef struct {
      double x, y, z;
      double mass;
   } Particle;

   typedef struct {
      double vx, vy, vz;
   } ParticleV;

   Particle  particles[MAX_PARTICLES];  /* Particles on all nodes */
   ParticleV vector[MAX_PARTICLES]; /* Particle velocity */
   int       counts[MAX_PROCS];         /* Number of ptcls on each proc */
   int       displacements[MAX_PROCS];  /* Offsets into particles */
   int       offsets[MAX_PROCS];        /* Offsets used by the master */
   int       particle_number, i, j,
             my_offset;                 /* Location of local particles */
   int       total_particles;           /* Total number of particles */
   int       count;                     /* Number of times in loop */

   MPI_Datatype particle_type;

   double    time;                      /* Computation time */
   double    dt, dt_old;                /* Integration time step */
   double    a0, a1, a2;
double    start_time, end_time;

MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &pool_size);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

if (my_rank == MASTER_RANK) i_am_the_master = TRUE;

particle_number = MAX_PARTICLES / pool_size;

if (i_am_the_master)
   printf ("%d particles per processor\n", particle_number);

MPI_Type_contiguous ( 4, MPI_DOUBLE, &particle_type );
MPI_Type_commit ( &particle_type );

MPI_Allgather ( &particle_number, 1, MPI_INT, counts, 1, MPI_INT,
                MPI_COMM_WORLD );

displacements[0] = 0;
for (i = 1; i < pool_size; i++)
   displacements[i] = displacements[i-1] + counts[i-1];
total_particles = displacements[pool_size - 1]
                  + counts[pool_size - 1];

if (i_am_the_master)
   printf ("total number of particles = %d\n", total_particles);

my_offset = displacements[my_rank];

MPI_Gather ( &my_offset, 1, MPI_INT, offsets, 1, MPI_INT, MASTER_RANK,
             MPI_COMM_WORLD );

if (i_am_the_master) {
   printf ("offsets: ");
   for (i = 0; i < pool_size; i++)
      printf ("%d ", offsets[i]);
   printf("\n");
}

srand48((long) my_rank);

for (i = 0; i < particle_number; i++) {
   particles[my_offset + i].x = drand48();
   particles[my_offset + i].y = drand48();
   particles[my_offset + i].z = drand48();
   particles[my_offset + i].mass = 1.0;
}

start_time = MPI_Wtime();

MPI_Allgatherv ( particles + my_offset, particle_number,
                 particle_type,
                 particles, counts, displacements, particle_type,
                 MPI_COMM_WORLD );

end_time = MPI_Wtime();
```

```
   if (i_am_the_master) {
      printf ("Communicating = %8.5f seconds\n", end_time - start_time);
      printf ("particles[offsets[i]].x: ");
      for (i = 0; i < pool_size; i++)
         printf ("%8.5f ", particles[offsets[i]].x);
      printf("\n"); fflush(stdout);
   }

   start_time = MPI_Wtime();

   count = 0;
   for (i = 0; i < particle_number; i++) {

      vector[my_offset + i].vx = 0.0;
      vector[my_offset + i].vy = 0.0;
      vector[my_offset + i].vz = 0.0;

      for (j = 0; j < total_particles; j++) {
         if (j != i) {

            double dx, dy, dz, r2, r, mimj_by_r3;

            dx = particles[my_offset + i].x - particles[j].x;
            dy = particles[my_offset + i].y - particles[j].y;
            dz = particles[my_offset + i].z - particles[j].z;

            r2 = dx * dx + dy * dy + dz * dz; r = sqrt(r2);

            if (r2 < EPSILON) mimj_by_r3 = 0.0;
            else
               mimj_by_r3 = particles[my_offset + i].mass
                            * particles[j].mass / (r2 * r);

            vector[my_offset + i].vx = vector[my_offset + i].vx +
                                       mimj_by_r3 * dx;
            vector[my_offset + i].vy = vector[my_offset + i].vy +
                                       mimj_by_r3 * dy;
            vector[my_offset + i].vz = vector[my_offset + i].vz +
                                       mimj_by_r3 * dz;
            count = count + 1;
         }
      }
   }

   end_time = MPI_Wtime();
   if (i_am_the_master)
      printf ("done my job in %8.5f seconds, waiting for slow \
processes...\n", end_time - start_time);

   MPI_Barrier (MPI_COMM_WORLD);

   end_time = MPI_Wtime();

   if (i_am_the_master)
         printf ("evaluated %d 3D interactions in %8.5f seconds\n",
                 count * pool_size, end_time - start_time);
   MPI_Finalize ();
}
```

And here is how this code compiles and runs:

```
gustav@sp20:../MPI 18:43:30 !588 $ mpcc -o particles particles.c -lm
gustav@sp20:../MPI 18:43:47 !589 $ cat particles.ll
# @ job_type = parallel
# @ environment = COPY_ALL; MP_EUILIB=ip; MP_INFOLEVEL=2
# @ requirements = (Adapter == "hps_ip") && (Machine != "sp43") && (Machine != "sp42")
# @ min_processors = 6
# @ max_processors = 10
# @ class = test
# @ output = particles.out
# @ error = particles.err
# @ executable = /usr/bin/poe
# @ arguments = particles
# @ queue
gustav@sp20:../MPI 18:43:52 !590 $ llsubmit particles.ll
submit: The job "sp20.123" has been submitted.
gustav@sp20:../MPI 18:43:59 !591 $ llq | grep gustav
gustav@sp20:../MPI 18:44:39 !592 $ cat particles.out
100 particles per processor
total number of particles = 1000
offsets: 0 100 200 300 400 500 600 700 800 900
Communicating =  0.00802 seconds
particles[offsets[i]].x:  0.17083  0.04163  0.91243  0.78323  0.65404  0.52484  0.39564  0.26644  0.13725
done my job in  0.09365 seconds, waiting for slow processes...
evaluated 999000 3D interactions in  0.09555 seconds
gustav@sp20:../MPI 18:44:43 !593 $
```

The messages that the program has written on standard output may not make that much sense to you right now, but they'll become clearer as we discuss the code in detail.

The Discussion

The program begins in the usual way: participating processes find out about the size of the process pool and their own place (rank, master) within that pool:

```
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &pool_size);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
   if (my_rank == MASTER_RANK) i_am_the_master = TRUE;
```

Then every process finds about the number of particles that it is going to look after:

```
   particle_number = MAX_PARTICLES / pool_size;
   if (i_am_the_master)
      printf ("%d particles per processor\n", particle_number);
```

Particles are described in terms of structures that comprise information about particle's mass and its position:

```
   typedef struct {
      double x, y, z;
      double mass;
   } Particle;
...
     Particle  particles[MAX_PARTICLES];  /* Particles on all nodes */
```

Because we are going to send particles between processes, we should define the corresponding `MPI_Datatypte` so that we can operate on particles as single entities, instead of having to split each of them into four real numbers and send those numbers separately or as an array.

In this case we rely on the fact that most C compilers implement a structure such as a `Particle` as four contigous double precision numbers. And MPI has a special way of handling such objects:

```
   MPI_Type_contiguous ( 4, MPI_DOUBLE, &particle_type );
   MPI_Type_commit ( &particle_type );
```

New MPI types, once defined, have to be *committed*. That operation allows MPI and, possibly, hardware to rearrange its internal communication buffers and links to handle the new type more efficiently. We have already encountered that operation in our previous example.

The next operation:

```
   MPI_Allgather ( &particle_number, 1, MPI_INT, counts, 1, MPI_INT,
                   MPI_COMM_WORLD );
```

gathers information about the number of particles every process is going to look after on an array `counts`. Because this is an All-Gather operation, every process ends up with data in its own copy of `counts`. The semantics of this operations is as follows: every process puts an integer number into its own slot called `particle_number`. The address of that slot, `&particle_number` points to the beginning of the send buffer. There is only 1 item of type `MPI_INT` in that buffer. Once the operation completes, the contribution from process 0 goes into `counts[0]`, the contribution from process 1 goes into `counts[1]`, and so on. Every process gets its own array `counts` filled. What goes into `counts` is one number of type `MPI_INT` from each process. It is actually possible to re-interpret the data that is sent on arrival, so that you can send `MPI_LONG` and receive it as 2 `MPI_INT` per slot of `counts`. In that case every slot of `counts` would have enough space for 2 integers.

Now every process evaluates the `displacements`, i.e., the positions within the array of particles where particles belonging to a given process begin:

```
   displacements[0] = 0;
   for (i = 1; i < pool_size; i++)
      displacements[i] = displacements[i-1] + counts[i-1];
   total_particles = displacements[pool_size - 1]
                     + counts[pool_size - 1];
```

Every process evaluates the total number of particles in the system too.

Observe that although we have given an equal number of particles to each process, the whole logic of the program is such that we can handle different numbers of particles per process.

It is now up to the master process to announce the total number of particles:

```
if (i_am_the_master)
   printf ("total number of particles = %d\n", total_particles);
```

And once this is done, every process evaluates the place in the array of particles where its own particles begin:

```
my_offset = displacements[my_rank];
```

This information is then communicated to the master process, who prints on standard output information about offsets corresponding to every process:

```
MPI_Gather ( &my_offset, 1, MPI_INT, offsets, 1, MPI_INT, MASTER_RANK,
             MPI_COMM_WORLD );
if (i_am_the_master) {
   printf ("offsets: ");
   for (i = 0; i < pool_size; i++)
      printf ("%d ", offsets[i]);
   printf("\n");
}
```

The operation `MPI_Gather` works like `MPI_Allgather`, but the difference is that only one process, called the *root* process does the collection and receives data from all other processes this time. The root process here is the `MASTER_RANK` process.

Now every process initializes positions of its own particles, only, to some random numbers, and sets their mass to 1:

```
srand48((long) my_rank);
for (i = 0; i < particle_number; i++) {
   particles[my_offset + i].x = drand48();
   particles[my_offset + i].y = drand48();
   particles[my_offset + i].z = drand48();
   particles[my_offset + i].mass = 1.0;
}
```

And here all processes exchange their particle data with other processes:

```
start_time = MPI_Wtime();
MPI_Allgatherv ( particles + my_offset, particle_number,
                 particle_type,
                 particles, counts, displacements, particle_type,
                 MPI_COMM_WORLD );

end_time = MPI_Wtime();
```

Function `MPI_Wtime()` returns wall clock time in seconds. This is a very useful MPI function, which is handy when timing MPI programs. Remember that to a user the only thing that really matter is the wall clock time, i.e., the time she's going to wait for her program to finish execution. CPU time is for clerks who administer supercomputer centres.

Now let us have a closer look at the `MPI_Allgatherv` operation. The operation works much like `MPI_Allgather`, but it allows for contributions of different length by different processes. The send buffer for a given process begins from `particles + my_offset`. Every process sends `particle_number` of objects whose type is `particle_type`. The data will be collected on `particles`, receiving `counts[i]` data items from process i, and once that data has been received it should be written on `particles` beginning from position `displacements[i]` within that array. The received data items are expected to be of type `particle_type`. Again observe that MPI allows for data to be cast on a different type while this operation is under way.

After this operation completes the master process writes on standard output a message about how long this communication operation took. Also, it writes x coordinates of particles located at the beginning of the sectors corresponding to different processes, just so that you can see that there are really some particles there.

Now we start out timer again and perform computation on particles:

```
start_time = MPI_Wtime();

count = 0;
for (i = 0; i < particle_number; i++) {

   vector[my_offset + i].vx = 0.0;
   vector[my_offset + i].vy = 0.0;
   vector[my_offset + i].vz = 0.0;

   for (j = 0; j < total_particles; j++) {
      if (j != i) {

         double dx, dy, dz, r2, r, mimj_by_r3;

         dx = particles[my_offset + i].x - particles[j].x;
         dy = particles[my_offset + i].y - particles[j].y;
         dz = particles[my_offset + i].z - particles[j].z;

         r2 = dx * dx + dy * dy + dz * dz; r = sqrt(r2);

         if (r2 < EPSILON) mimj_by_r3 = 0.0;
         else
            mimj_by_r3 = particles[my_offset + i].mass
                       * particles[j].mass / (r2 * r);

         vector[my_offset + i].vx = vector[my_offset + i].vx +
                                    mimj_by_r3 * dx;
         vector[my_offset + i].vy = vector[my_offset + i].vy +
                                    mimj_by_r3 * dy;
         vector[my_offset + i].vz = vector[my_offset + i].vz +
                                    mimj_by_r3 * dz;
            count = count + 1;
         }
      }
   }
```

For every particle i within every process' region of responsibility, the velocity of that particle $\boldsymbol{v}_i$ is set to 0. Then for that particle we evaluate forces with which all other particles (j) in the entire system act on it:

$$\begin{aligned} \boldsymbol{F}_{ij} &= \frac{m_i m_j}{(\boldsymbol{r}_i - \boldsymbol{r}_j)^2} \frac{\boldsymbol{r}_i - \boldsymbol{r}_j}{|\boldsymbol{r}_i - \boldsymbol{r}_j|} \\ \boldsymbol{v}_i &\leftarrow \boldsymbol{v}_i + \boldsymbol{F}_{ij} \end{aligned}$$

and then the effect that those forces have on particle i's velocity.

The computation here is just a fake, and it's purpose is merely to demonstrate that something gets computed. Our laws of dynamics are a little strange, it is not entirely clear if m_i is really a mass or a charge, Δt is set to 1, and if m_i is a mass, then we quite unnecessarily multiply m_i by m_j, because then we should really divide it all back by m_i. But who cares. For the sake of this program the most important thing is to measure how long this computation will take and compare that with the time used by the communication operations.

As soon as the master process has finished its own work, it writes on standard output how long it spent computing forces and advancing velocities.

```
   end_time = MPI_Wtime();
   if (i_am_the_master)
      printf ("done my job in %8.5f seconds, waiting for slow \
processes...\n", end_time - start_time);
```

Some processes may have taken longer to do that than others. So we set up a barrier.

```
   MPI_Barrier (MPI_COMM_WORLD);
```

This is a synchronization device. The function returns *only* when *all* processes within the `MPI_COMM_WORLD` communicator, in this case, have *joined* the barrier. So now the master can measure time again, and tell us how long it took for the slowest process to finish the job:

```
   end_time = MPI_Wtime();

   if (i_am_the_master)
         printf ("evaluated %d 3D interactions in %8.5f seconds\n",
                 count * pool_size, end_time - start_time);
```

And this is it.

Let us now have a look again at the output produced by the program:

```
100 particles per processor
total number of particles = 1000
offsets: 0 100 200 300 400 500 600 700 800 900
```

```
Communicating =  0.00802 seconds
particles[offsets[i]].x:  0.17083  0.04163  0.91243  0.78323  0.65404  0.52484  0.39564  0.26644  0.13725  0.00805
done my job in  0.09365 seconds, waiting for slow processes...
evaluated 999000 3D interactions in  0.09555 seconds
```

The most important thing to observe is that on this example it took only 0.008 seconds to exchange information using `MPI_Allgatherv`, whereas it took 0.095 seconds to perform the computation. Our program spends only 8% of its wall clock time communicating which is quite good.

Will this work just as well if we increase the number of particles to, say, 10,000? It will work even better:

```
gustav@sp20:../MPI 21:21:17 !575 $ cat particles.out
1111 particles per processor
total number of particles = 9999
offsets: 0 1111 2222 3333 4444 5555 6666 7777 8888
Communicating =  0.03104 seconds
particles[offsets[i]].x:  0.17083  0.04163  0.91243  0.78323  0.65404  0.52484  0.39564  0.26644  0.13725
done my job in 10.61837 seconds, waiting for slow processes...
evaluated 99970002 3D interactions in 20.73818 seconds
gustav@sp20:../MPI 21:21:22 !576 $
```

This time our communication took only 0.15% of computation time.

The moral of the story is that in this case, at least, the larger the size of the problem on a node the more efficient the parallelisation, because the cost of communication is going to be, in proportion, that much less. This state of affairs is pretty common, meaning that the larger the problem in general the more effective parallelisation may be. For smaller problems on the other hand, the cost of parallelisation may be sometimes prohibitive: a parallelised version of a small problem may take much longer to execute than a sequential version.

8.3.3 Manipulating Communicators

So far we haven't done any explicit manipulation of communicators. In our example program that set up a communication framework for finite difference codes (e.g., a Laplace solver) we have used a powerful wrapper, `MPI_Cart_create`, that took care of all details associated with creating a new communicator.

In the following code we're going to do this ourselves, although the new communicator will not have a topology associated with it.

The general synopsis for the code that follows is simple. We're going to have one process responsible for generating sequences of random numbers. The remaining processes, including the one responsible for I/O, will form a new communicator within which computations on those sequences will be carried on.

Having received a sequence from the random number generator a worker process is going to convert those to random x and y coordinates within a square $[-1,1]\times[-1,1]$. Some points generated thusly may live outside a circle with its centre on $(0,0)$ and radius 1, and some may live inside it. If a point is within the circle we add 1 to the `in` counter, otherwise we add 1 to the `out` counter. The number of randomly generated points that fall into the circle is going to be proportional to π, whereas the number of points that fall outside the circle is going to be proportional to $4 - \pi$, with the total number of points being proportional to 4, because the surface area of a square $[-1, 1] \times [-1, 1]$ is 4, and the surface area of a circle of radius 1 is π. Therefore $4 \times \texttt{in} / (\texttt{in} + \texttt{out}) \approx \pi$.

Every process can evaluate this on its own. But if they were to pool their own results, we would end up having a lot more data and thus our evaluation of π would be that much more accurate.

The Code

Here is the code:

```
#include <stdio.h>
#include <math.h>
#include "mpi.h"
#define CHUNKSIZE 1000
#define REQUEST      1
#define REPLY        2

main(argc, argv)
     int argc;
     char *argv[];
{
  int iter;
  int in, out, i, iters, max, ix, iy, ranks[1], done, temp;
  double x, y, Pi, error, epsilon;
  int numprocs, myid, server, totalin, totalout, workerid;
  int rands[CHUNKSIZE], request;
  MPI_Comm world, workers;
  MPI_Group world_group, worker_group;
  MPI_Status stat;

  MPI_Init(&argc, &argv);
  world = MPI_COMM_WORLD;
  MPI_Comm_size(world, &numprocs);
  MPI_Comm_rank(world, &myid);
  server = numprocs - 1;
  if (myid == 0)
    sscanf( argv[1], "%lf", &epsilon);
  MPI_Bcast(&epsilon, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Comm_group( world, &world_group);
  ranks[0] = server;
  MPI_Group_excl(world_group, 1, ranks, &worker_group);
  MPI_Comm_create(world, worker_group, &workers);
  MPI_Group_free(&worker_group);

  if(myid == server) {
    do {
      MPI_Recv(&request, 1, MPI_INT, MPI_ANY_SOURCE, REQUEST, world, &stat);
      if (request) {
        for (i = 0; i < CHUNKSIZE; i++)
          rands[i] = random();
        MPI_Send(rands, CHUNKSIZE, MPI_INT, stat.MPI_SOURCE, REPLY, world);
      }
    }
    while (request> 0);
  }
  else {
    request = 1;
    done = in = out = 0;
    max = INT_MAX;
    MPI_Send(&request, 1, MPI_INT, server, REQUEST, world);
    MPI_Comm_rank(workers, &workerid);
    iter = 0;
    while(!done) {
      iter++;
      request = 1;
      MPI_Recv(rands, CHUNKSIZE, MPI_INT, server, REPLY, world, &stat);
      for (i=0; i < CHUNKSIZE; ) {
        x = (((double) rands[i++])/max) * 2 - 1;
        y = (((double) rands[i++])/max) * 2 - 1;
        if (x*x + y*y < 1.0)
          in++;
        else
          out++;
      }
      MPI_Allreduce(&in, &totalin, 1, MPI_INT, MPI_SUM, workers);
      MPI_Allreduce(&out, &totalout, 1, MPI_INT, MPI_SUM, workers);
      Pi = (4.0*totalin)/(totalin + totalout);
      error = fabs( Pi-3.141592653589793238462643);
      done = ((error < epsilon) || ((totalin+totalout) > 1000000));
      request = (done) ? 0 : 1;
      if (myid == 0) {
        printf( "\rpi = %23.20lf", Pi);
        MPI_Send(&request, 1, MPI_INT, server, REQUEST, world);
      }
      else {
        if (request)
          MPI_Send(&request, 1, MPI_INT, server, REQUEST, world);
      }
    }
    MPI_Comm_free(&workers);
  }
  if (myid == 0)
    printf( "\npoints: %d\nin: %d, out: %d\n",
            totalin+totalout, totalin, totalout);
  MPI_Finalize();
}
```

And here is how to compile and run this code:

```
gustav@sp20:../MPI 17:39:37 !550 $ mpcc -o pirand pirand.c
gustav@sp20:../MPI 17:39:50 !551 $ cat pirand.ll
# @ job_type = parallel
# @ environment = COPY_ALL; MP_EUILIB=ip; MP_INFOLEVEL=2
# @ requirements = (Adapter == "hps_ip")
# @ min_processors = 6
# @ max_processors = 10
# @ class = test
# @ output = pirand.out
# @ error = pirand.err
# @ executable = /usr/bin/poe
# @ arguments = pirand
# @ queue
gustav@sp20:../MPI 17:39:57 !552 $ llsubmit pirand.ll
submit: The job "sp20.135" has been submitted.
gustav@sp20:../MPI 17:40:14 !553 $ cat pirand.out
pi =  3.13959541604384639868
points: 1003500
in: 787646, out: 215854
gustav@sp20:../MPI 17:41:30 !554 $
```

You can also invoke program `pirand` with an option that specifies the value of the ε parameter, which is used to assess the accuracy of the computation. Let us try to do this interactively a few times:

```
gustav@sp20:../MPI 17:41:30 !554 $ poe pirand 0.001 -procs 8
pi =  3.14158730158730170601
points: 31500
in: 24740, out: 6760
gustav@sp20:../MPI 17:43:32 !555 $ poe pirand 0.00001 -procs 8
pi =  3.14158730158730170601
points: 31500
in: 24740, out: 6760
gustav@sp20:../MPI 17:43:46 !556 $ poe pirand 0.000001 -procs 8
pi =  3.13962437562437557403
points: 1001000
in: 785691, out: 215309
gustav@sp20:../MPI 17:44:13 !557 $ poe pirand 0.01 -procs 8
pi =  3.15085714285714280081
points: 3500
in: 2757, out: 743
gustav@sp20:../MPI 17:44:28 !558 $
```

Observe that asking for too good a resolution does not pay. The program runs up to the preset limit of one million points and returns a result that is less accurate than the result returned with only 31,500 points, which is accurate to within 0.00001. This tells us something about the random number generator, which, clearly, isn't random enough for long sequences.

The Discussion

Let us now have a closer look at the code and see what happens there.

The code begins with the usual incantations: `MPI_Init`, `MPI_Comm_size`, `MPI_Comm_rank`. But this time we have two special processes. Process with rank 0, which, as usual, is responsible for I/O, and then process rank `numprocs - 1`, which we're going to call a `server`.

```
  MPI_Init(&argc, &argv);
  world = MPI_COMM_WORLD;
  MPI_Comm_size(world, &numprocs);
  MPI_Comm_rank(world, &myid);
  server = numprocs - 1;
```

Process rank 0 inspects the command line in order to find the value of ϵ, which is going to be compared to $\pi - \pi'$, where π' is going to be the approximate value of π as evaluated by our Monte-Carlo computation. The value is then broadcast to all other processes:

```
if (myid == 0)
  sscanf( argv[1], "%lf", &epsilon);
MPI_Bcast(&epsilon, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Now we commence the formation of the new communicator:

```
MPI_Comm_group( world, &world_group);
ranks[0] = server;
MPI_Group_excl(world_group, 1, ranks, &worker_group);
MPI_Comm_create(world, worker_group, &workers);
MPI_Group_free(&worker_group);
```

In order to construct the new communicator we must extract a group of processes associated with the original communicator, which is referred to here as `world`. Then we can perform an operation on that group. For example we can exclude some processes from it, or we could add some processes to it using `MPI_Group_incl`, or if we had two groups we could find their intersection using `MPI_Group_intersection`, or we could form a union of such groups using `MPI_Group_union`. Once we have formed a new group of processes we can associate a new communicator with them by calling `MPI_Comm_create`. Once we have created that new communicator, we no longer need to keep the group that the communicator is associated with around, so we dispose of it by calling `MPI_Group_free`.

The synopsis for `MPI_Comm_group` is as follows: the first argument is a communicator, the second argument is a pointer to a container that is going to hold a group of processes associated with the communicator once the function returns.

The synopsis for `MPI_Group_excl`: the first argument is a group from which processes are to be excluded, then we have to specify how many processes are to be excluded, and then we have to pass an array, whose entries correspond to the rank number of the excluded processes. The last argument is a pointer to a container that will hold the new group once the function returns.

The synopsis for `MPI_Comm_create`: here the first argument is the original communicator, the second argument is a group of processes within which the new communicator is to be associated, and the third argument is a pointer to a container that will hold the new communicator once the function returns.

Having excluded the random number `server` from the pool we can subdivide our program into a server part and a worker part.

The server part is simple:

```
if(myid == server) {
  do {
    MPI_Recv(&request, 1, MPI_INT, MPI_ANY_SOURCE, REQUEST, world, &stat);
    if (request) {
      for (i = 0; i < CHUNKSIZE; i++)
```

```
        rands[i] = random();
      MPI_Send(rands, CHUNKSIZE, MPI_INT, stat.MPI_SOURCE, REPLY, world);
    }
  }
  while (request> 0);
}
```

The server waits for a `request`, which is going to be a single integer number. The tag of this message is going to be `REQUEST`, and the message is expected to arrive from `MPI_ANY_SOURCE` within the `world` communicator. If the integer number sent in `request` is greater than zero then the server generates a `CHUNKSIZE` of random integer numbers between 0 and `max`, which is equal to `INT_MAX`, and the latter is a system constant defined in `/usr/include/sys/limits.h`, which in turn is included by `/usr/include/stdio.h`. The array `rands` filled with `CHUNKSIZE` of random integers is then sent back to process `stat.MPI_SOURCE`, which is the same process that submitted the original request.

If the value of `request` is less than 1, then the server process terminates.

Now let us have a look at what the worker processes are going to do at the same time.

Their life begins with some initializations that by now should be pretty obvious:

```
request = 1;
done = in = out = 0;
max = INT_MAX;
```

And then each worker sends a request to the random number server and finds about its own rank within the `workers` communicator. No receive is attempted at this stage:

```
MPI_Send(&request, 1, MPI_INT, server, REQUEST, world);
MPI_Comm_rank(workers, &workerid);
```

Now the workers enter the main loop:

```
iter = 0;
while(!done) {
  iter++;

  ...

  error = fabs( Pi-3.141592653589793238462643);
  done = ((error < epsilon) || ((totalin+totalout) > 1000000));
  request = (done) ? 0 : 1;
  if (myid == 0) {
    printf( "\rpi = %23.20lf", Pi);
    MPI_Send(&request, 1, MPI_INT, server, REQUEST, world);
  }
  else {
    if (request)
      MPI_Send(&request, 1, MPI_INT, server, REQUEST, world);
  }
}
```

Within this loop some computation and exchange of information is done. We'll focus on that part later. After every process has some idea about what its value of π' is, it is compared to the exact value of π and the absolute value of the difference is written on `error`. The we check if that error is less than ε or if we have exceeded a maximum number of points, for all processes taken together. If any of this conditions is satisfied, even if π' has not been evaluated with a sufficient accuracy yet, we flag the job as `done`, and the `request` is set to 0 or to 1 otherwise. It befalls to the *speaker* process, i.e., to the guy whose rank number within the `world` communicator is 0, to send that terminating message to the server. But the *speaker* process sends some message always: it is simply a request. The other processes, i.e., the workers who are not the *speaker* send a request only if it's kosher.

Now let us have a look at the computation itself.

We begin by receiving a sequence of random integers from the server. The package is of length `CHUNKSIZE`, the data items are all of type `MPI_INT`, the sender is `server` and the tag is `REPLY`. The communication takes place within the `world` communicator, because the server does not belong to the `workers` communicator:

```
MPI_Recv(rands, CHUNKSIZE, MPI_INT, server, REPLY, world, &stat);
```

Now we convert the whole sequence to points (x, y), check which of those are within the circle of radius 1 ($x^2 + y^2 < 1$) and which are without and increment appropriate counters (`in` and `out`).

```
for (i=0; i < CHUNKSIZE; ) {
  x = (((double) rands[i++])/max) * 2 - 1;
  y = (((double) rands[i++])/max) * 2 - 1;
  if (x*x + y*y < 1.0)
    in++;
  else
    out++;
}
```

Why is `rands[i++]/max` multiplied by 2 and then 1 subtacted from the result? Function random returns pseudo-random numbers between 0 and $2^{31} - 1$. `INT_MAX` is $2^{31} - 1$. So, the largest x or y is going to be 1 and the smallest is going to be -1.

The next step is to exchange our own local `in` and `out` with all other worker processes. This is done by the two operations:

```
MPI_Allreduce(&in, &totalin, 1, MPI_INT, MPI_SUM, workers);
MPI_Allreduce(&out, &totalout, 1, MPI_INT, MPI_SUM, workers);
```

These are not ordinary reduce operations. They are *all*-reduce operations. The work like a normal reduce, but instead of the result being known only to the root process, here it is known to all processes. So every worker process now ends up with a sum of all `in`s in its `totalin` and the sum of all `out`s in its `totalout`. This communication takes place only within the `workers` communicator, so that the random number server process is not bothered.

Finally π can be evaluated thusly:

```
    Pi = (4.0*totalin)/(totalin + totalout);
```

This is not the end of what the workers processes have to do. After they have finished all their work, they must release the communicator. This is done by calling:

```
  MPI_Comm_free(&workers);
```

If you look at this program in “Using MPI” by Gropp, Lusk, and Skjellum, you'll find that `MPI_Comm_free(&workers)` is called just ahead of `MPI_Finalize` and outside of the

```
if(myid == server) {
   ...
else {
   ...
}
```

clause. This is a bug, because this means that the server process which does not belong to the `workers` communicator will call `MPI_Comm_free(&workers)` too. But for that process `workers` is going to evaluate to `MPI_COMM_NULL`, so the call will really be `MPI_Comm_free(&MPI_COMM_NULL)`. But `MPI_COMM_NULL` is just a literal constant, and it doesn't have any address, so the whole hell is gonna break loose.

The last step before quitting the program is to write a message about the total number of points processed and generated by the program. This is done by the *speaker* process:

```
if (myid == 0)
  printf( "\npoints: %d\nin: %d, out: %d\n",
          totalin+totalout, totalin, totalout);
```

8.4 Fortran Interface

MPI was defined from the beginning with C and F77 interfaces. The latter can be used without any change in F90 programs. The basic difference between C and Fortran interfaces is that in Fortran all MPI calls are implemented as subroutines, not functions. The error code is then returned in the last parameter, with other parameters being the same as in C. Some names are changed, e.g., instead of `MPI_CHAR` we use `MPI_CHARACTER` in fortran.

The following simple examples illustrate how to use F90 interfaces on the SP.

The MPI wraper for Fortran-90 codes on the SP is called `mpxlf90`. It works much the same as `mpcc`. Instead of

```
#include "mpi.h"
```

in Fortran you must use

```
INCLUDE 'mpif.h'
```

INCLUDE isn't a dinkum Fortran keyword. There should really be a compiled interface, which could then be USEd.

8.4.1 Hello World

```
gustav@sp21:../MPI 18:32:17 !622 $ cat hello-1.f
PROGRAM hello_world

  INCLUDE "mpif.h"

  INTEGER :: ierror, resultlen
  CHARACTER(len=256) :: name = " "

  CALL mpi_init(ierror)
  CALL mpi_get_processor_name(name, resultlen, ierror)
  WRITE (*,*) TRIM(name), ': hello world'
  CALL mpi_finalize(ierror)

END PROGRAM hello_world
gustav@sp21:../MPI 18:32:21 !623 $ mpxlf90 -o hello-1 hello-1.f
** hello_world   === End of Compilation 1 ===
1501-510  Compilation successful for file hello-1.f.
gustav@sp21:../MPI 18:32:49 !624 $ poe hello-1 -procs 8
 sp17.ucs.indiana.edu: hello world
 sp19.ucs.indiana.edu: hello world
 sp20.ucs.indiana.edu: hello world
 sp22.ucs.indiana.edu: hello world
 sp23.ucs.indiana.edu: hello world
 sp24.ucs.indiana.edu: hello world
 sp40.ucs.indiana.edu: hello world
 sp21.ucs.indiana.edu: hello world
gustav@sp21:../MPI 18:33:06 !625 $
```

8.4.2 Greetings, Master

```
gustav@sp21:../MPI 18:33:06 !625 $ cat hello-2.f
PROGRAM hello_master

  IMPLICIT NONE

  INCLUDE "mpif.h"

  INTEGER :: ierror, result_len, pool_size, count, my_rank
  INTEGER, PARAMETER :: master_rank = 0, bufsiz = 512
  INTEGER, DIMENSION(MPI_STATUS_SIZE) :: status
  CHARACTER(len=bufsiz) :: my_name = " ", master_name = " ", &
       send_buffer = " ", my_rank_string = " ", recv_buffer = " "
  LOGICAL :: i_am_the_master = .FALSE.

  CALL mpi_init(ierror)
  CALL mpi_comm_size(MPI_COMM_WORLD, pool_size, ierror)
  CALL mpi_comm_rank(MPI_COMM_WORLD, my_rank, ierror)
  CALL mpi_get_processor_name(my_name, result_len, ierror)

  IF (my_rank .EQ. master_rank) THEN
     i_am_the_master = .TRUE.
     master_name = my_name
  END IF

  CALL mpi_bcast(master_name, bufsiz, MPI_CHARACTER, master_rank, &
       MPI_COMM_WORLD, ierror)

  WRITE(my_rank_string, '(1i3)') my_rank
  send_buffer = 'hello ' // TRIM(master_name) &
       // ', greetings from ' // my_name // ', rank = ' &
       // my_rank_string
  CALL mpi_send(send_buffer, bufsiz, MPI_CHARACTER, master_rank, 0, &
       MPI_COMM_WORLD, ierror)

  IF (i_am_the_master) THEN
     DO count = 1, pool_size
        CALL mpi_recv (recv_buffer, bufsiz, MPI_CHARACTER, MPI_ANY_SOURCE, &
             MPI_ANY_TAG, MPI_COMM_WORLD, status, ierror)
        WRITE(*,*) TRIM(recv_buffer)
     END DO
  END IF

  CALL mpi_finalize(ierror)

END PROGRAM hello_master
gustav@sp21:../MPI 18:34:34 !626 $ mpxlf90 -o hello-2 hello-2.f
** hello_master   === End of Compilation 1 ===
1501-510  Compilation successful for file hello-2.f.
gustav@sp21:../MPI 18:34:46 !627 $ poe hello-2 -procs 8
 hello sp17.ucs.indiana.edu, greetings from sp17.ucs.indiana.edu
 hello sp17.ucs.indiana.edu, greetings from sp22.ucs.indiana.edu
 hello sp17.ucs.indiana.edu, greetings from sp20.ucs.indiana.edu
 hello sp17.ucs.indiana.edu, greetings from sp19.ucs.indiana.edu
 hello sp17.ucs.indiana.edu, greetings from sp24.ucs.indiana.edu
 hello sp17.ucs.indiana.edu, greetings from sp21.ucs.indiana.edu
 hello sp17.ucs.indiana.edu, greetings from sp23.ucs.indiana.edu
 hello sp17.ucs.indiana.edu, greetings from sp40.ucs.indiana.edu
gustav@sp21:../MPI 18:35:02 !628 $
```

8.4.3 Dividing the Pie

```
gustav@sp21:../MPI 19:14:39 !667 $ cat pie.f
PROGRAM pie

  IMPLICIT NONE
  INCLUDE 'mpif.h'

  INTEGER :: ierror, pool_size, my_rank, n, i
  INTEGER, PARAMETER :: master_rank = 0
  LOGICAL :: i_am_the_master = .FALSE.
  REAL(kind=8) :: h, sum, x, my_pi, pi

  CALL mpi_init(ierror)
  CALL mpi_comm_size(MPI_COMM_WORLD, pool_size, ierror)
  CALL mpi_comm_rank(MPI_COMM_WORLD, my_rank, ierror)

  IF (my_rank == master_rank) i_am_the_master = .TRUE.

  IF (i_am_the_master) THEN
     WRITE(*,*) 'Enter the number of intervals: '
     READ(*,*) n
     IF (n .EQ. 0) n = 100
  END IF

  CALL mpi_bcast(n, 1, MPI_INTEGER, master_rank, MPI_COMM_WORLD, ierror)

  h = 1.0_8 / n
  sum = 0.0_8
  DO i = my_rank + 1, n, pool_size
     x = h * (i - 0.5_8)
     sum = sum + f(x)
  END DO
  my_pi = h * sum

  CALL mpi_reduce(my_pi, pi, 1, MPI_DOUBLE_PRECISION, MPI_SUM, &
       master_rank, MPI_COMM_WORLD, ierror)

  IF (i_am_the_master) WRITE(*,*) 'pi is approximately ', pi

  CALL mpi_finalize(ierror)

CONTAINS

  FUNCTION f(x)
    REAL(kind=8) :: f, x

    f = 4.0_8 / (1.0_8 + x * x)

  END FUNCTION f

END PROGRAM pie
gustav@sp21:../MPI 19:15:16 !668 $ mpxlf90 -o pie pie.f
** pie   === End of Compilation 1 ===
1501-510  Compilation successful for file pie.f.
gustav@sp21:../MPI 19:15:39 !669 $ poe ./pie -procs 8
 Enter the number of intervals:
100000
 pi is approximately  3.14159265359813444
gustav@sp21:../MPI 19:16:02 !670 $
```

8.4.4 Exercise

Translate program "Bank Queue", section 8.2.4, page 202, from C to Fortran.

hints

1. There is no `flush` in Fortran, but you can use a utility routine from module `xlfutility` called `flush_`. See "XL Fortran for AIX, Lan-

guage Reference, Version 4, Release 1", pages 471 and 474 for more information. You can view this manual with `ghostview`. It lives on

`/afs/ovpit.indiana.edu/common/www/htdocs/gustav/SP-docs/xlf/xlflr.ps`

2. Use the same manual as a reference regarding any other Fortran queries.
3. Check carefully every MPI call against the on-line documentation for correct order of parameters, and especially `ierror`.
4. If you forget to put `ierror` in `CALL mpi_init(ierror)` your MPI program will crash instantaneously. Similarly remember to put `ierror` in `CALL mpi_finalize(ierror)`.

8.5 MPI Analysis and Profiling Tools

Consider the following simple HPF program:

```
program jacobi

  implicit none

!hpf$ nosequence

  integer, parameter :: n = 40, iterations = 1000
  integer, dimension(n), parameter :: north_boundary = 1, &
       east_boundary = 40, west_boundary = 40, south_boundary = 70
  integer, dimension(n, n) :: field = 3
  logical, dimension(n, n) :: mask = .true.
  integer :: i

!hpf$ align mask(:,:) with field
!hpf$ distribute (*, block) :: field

  field(ubound(field, dim=1), :) = east_boundary
  mask(ubound(mask, dim=1), :) = .false.
  field(lbound(field, dim=1), :) = west_boundary
  mask(lbound(mask, dim=1), :) = .false.
  field(:, ubound(field, dim=2)) = north_boundary
  mask(:, ubound(mask, dim=2)) = .false.
  field(:, lbound(field, dim=2)) = south_boundary
  mask(:, lbound(mask, dim=2)) = .false.

  call print_matrix(field)

  do i = 1, iterations
     where (mask)
        field = (eoshift(field, 1, dim=1) + eoshift(field, -1, dim=1) &
             + eoshift(field, 1, dim=2) + eoshift(field, -1, dim=2)) * 0.25
     end where
  end do

  call print_matrix(field)

contains
```

```
  subroutine print_matrix(field)
    integer, dimension(:,:) :: field
    integer :: i
    write(*, '(1x)')
    do i = size(field, dim=1), 1, -1
       write(*, '(1x, 40i3)') field(:,i)
    end do
  end subroutine print_matrix

end program jacobi
```

This is the simple Jacobi iteration program we have worked on in P573. I have changed the size of the matrix to 40×40.

Compile this program on the SP as follows:

```
gustav@sp20:../jacobi 17:14:24 !520 $ xlhpf90 -g -o jacobi jacobi.f
** jacobi   === End of Compilation 1 ===
1501-510  Compilation successful for file jacobi.f.
gustav@sp20:../jacobi 17:14:31 !521 $
```

Observe the presence of the `-g` switch, which adds debugging information to the binary.

We will now run this program with *tracing* turned on. There are two ways to do that. One is to define

```
$ export MP_TRACELEVEL=9
```

in your environment and then run the program under `poe`. The other way is to invoke `poe` with the `-tracelevel 9` option. If you wish to run the program under the LoadLeveler submit the following LoadLeveler job description file:

```
gustav@sp20:../jacobi 17:14:31 !521 $ cat jacobi.ll
# @ job_type = parallel
# @ environment = COPY_ALL; MP_EUILIB=us; MP_INFOLEVEL=6; MP_TRACELEVEL=9
# @ requirements = (Adapter == "hps_user")
# @ min_processors = 4
# @ max_processors = 8
# @ output = jacobi.out
# @ error = jacobi.err
# @ executable = /usr/bin/poe
# @ arguments = jacobi
# @ notification = always
# @ class = test
# @ queue
gustav@sp20:../jacobi 17:18:17 !522 $ llsubmit jacobi.ll
submit: The job "sp20.188" has been submitted.
gustav@sp20:../jacobi 17:18:29 !523 $
```

Observe that this time I have requested that the communication should take place through the *user space* rather than through TCP/IP. On P2SC nodes this is a more efficient way of transmitting messages between processors. On the so called *Silver Nodes*, and on the new Power-3 nodes, this is no longer the case, apparently.

When the job completes, apart from the usual `jacobi.err` and `jacobi.out` files, there should be a new file in your working directory, called `jacobi.trc`. This is the trace file.

We can now look at it with the visualisation tool, `vt`.

```
gustav@sp20:../jacobi 17:21:58 !528 $ vt -tracefile jacobi.trc &
[1] 25230
gustav@sp20:../jacobi 17:23:34 !529 $
```

When `vt` comes up it will flash a window saying:

```
      Postprocessing, please wait...
```

This may take a while, because the trace file in this case is nearly 17MB. When `vt` has finished post processing the trace file, it will say so. At this stage we can begin looking at the program.

`vt` will bring up two main windows. The first one contains a nowadays familiarly looking pushbuttons for `play`, `step`, `loop`, `reset`, and `stop`. There is also a speed scrollbar on the right and a `Tracefile Time Control` at the bottom.

The second window contains multiple push buttons for selecting various views. The view that you want to select initially is `Communication/Program`. Press this button. The window that comes up will be initially black. Now press the play button and watch the action unfold.

You will see a number of "thermometer" displays. Initially these will scroll slowly, without much action at all, until eventually you should see stright lines appear, which connect various points on those "thermometers".

Every "thermometer" represents an MPI process. There are fields of various colours on those "thermometers". The colours represent various activities. You can point at any particular field and click the left mouse button to see what a given processor was doing at this stage. If you click on the gray field a small window will pop up telling you:

```
processor 4:0: No Communications
```

If you click on the blue field, the pop up window will say:

```
processor 3:0: MPI Wait
```

And if you click on the pink field, the pop up window will say:

```
processor 5:0: MPI Immediate Receive
```

Looking at some parts of the parallel program, as it unfolds, you can see that there is a lot of `MPI Wait` in it. The white lines connecting the "thermometers" show which processes have been connected by message pipes at this stage.

If you right-click on the `Interprocessor Communication` window you will get another menu. There you can select various options, such as `Search`, `Parameters`, and `Configuration`. Go to `Parameters`. Here you will see what it is that the colours in the "thermometers" correspond to. The default is `Communication`. But if you left-click on the `keySpectrum` window, you can change that to `Random`, `Fade`, `Monochrome`, `Discrete`, `Continuous`, and `CPU Load`.

Now go back to the `VT View Selector` window and select another view, for example, `Message Status Matrix`. Rewind the trace, and replay it again.

The `Message Status Matrix` is a matrix, $n \times n$, where n is the number of processes. When there is a communication from, say, process 3 to process 5, then a square that corresponds to position $(3, 5)$ in the matrix lights up – in my case, actually, it goes black. You can change that, by right-clicking with your mouse on the matrix window, and selecting `Parameters` in the little pop-up menu.

Observe that for this program messages appear initially over the whole body of the matrix, but eventually they all fill just the last column of the matrix. When you get to this stage, stop the time scroll, and then left-click on any of the lit-up squares. You should see a message in a pop-up window similar to this one:

```
17:18:39.048969608
Messages sent from 1 to 7:1
MessageLength=800
CummLength=800
```

The fact that it is only the last column that is filled means that all messages are *gathered* by process rank 7, which in this case must have been made responsible for coordinating the work of the Jacobi iterator.

Another push button in the `VT View Selector` window called `Connectivity Graph` displays a similar information. But this time processes are represented by dots placed on the perimeter of a circle. Arcs connecting the dots then appear, as messages begin to fly between processes. As processes perform various actions, e.g., `No Communications`, or `MPI Blocking Send` the dots that correspond to those processes change colours. You can stop the trace replay, at any stage, and left-click on any of the dots, to see what it has been up to at that instance.

You can also select `Source Code` in the `VT View Selector` panel. As the replay proceeds, code lines that are being executed right now are highlighted.

8.5.1 Exercise 1

Recompile program "Bank Queue" with the `-g` switch, and run it with tracing turned on. Invoke the `vt` on the trace generated thusly.

Invoke the `Source Code` window and the `Interprocessor Communication` Window.

Now `Reset` the trace replay to the beginning and begin to step through the code manually by pressing the `Step` key in the main `Visualization Tool` panel. Observe that as you step in time you can see which process executes which line in the `Source Code` window, as the little highlighted rectangles that correspond to the participating processes descend on the source code lines from the top bar.

If you would like to scroll through the whole trace automatically, for this program you should slow down the trace replay to a minimum, because it is a very short program.

As you replay the trace at slow speed, stop it approximately in the middle. The `Interprocessor Communication` window looks pretty crowded. You can stretch the time axis in this window by changing the magnification in the main `vt` panel. Observe that for this program and with these parameters the processes involved spend most of their time waiting on `MPI Blocking Receive`. You can see that by right-clicking on the pink fields in the `Interprocessor Communication` window. Process 0 is not too busy either spending most time on `MPI Blocking Send` (blue fields).

It may happen that one or more of your processes are very slow compared to other processes and don't participate in the communication and computation at all. As I look at my display, I can see that process 5 hasn't done anything, until the very end when every other process finished its job already. In fact, after the whole computation has ended, I can see that process 0 still has to wait a very long time on `MPI Blocking Receive` for process 5, which owes it the result of multiplication of the row of matrix $\boldsymbol{A}$ it has received from it at the very beginning by vector $\boldsymbol{b}$.

Tools such as the `vt` are very enlightening. They show us how exorbitant is the cost of interprocess communication. This cost can be offset only by a very large amount of work dumped onto every slave process by the master.

Now let us have a look at some other display windows.

Click on the `User Utilization` button in the `VT View Selector` panel. By default you get a cumulative view. But if you invoke a display specific menu by right-clicking on the `User Utilization` window, you can select `Individual View` that will show you the history of the load you have subjected every CPU participating in the computation and in the communication to.

Looking at these diagrams it may appear that the program has used the system very effectively, because `User Utilization` is high, and `Processor Idle` is low. But remember that when you send messages to other processes, the CPUs involved are very busy doing all that, so the CPUs work just as hard when they send and receive messages, as when they do computations for you. If a CPU has to *wait* for a message, it spins and and checks various handles and buffers every now and then. This is all work, and this is going to show up as such in these diagrams. But it is not useful work from our point of view, because no computation is being done when the CPU spins awaiting a message.

As you scroll the `VT View Selector` window you can see that there is a lot of information available for you in the trace. But for the parallel programmer the most important information is contained in the `Communication/Program` group.

8.5.2 Exercise 2

Use the Visualization Tool, `vt`, to inspect communication patterns for other MPI programs we have discussed so far. In particular have a look at the `Interacting Particles` program. Vary the total number of particles in the program. Can you see the effect that increasing the size of the problem has on the efficacy of parallelization?

8.6 Parallel Debugging

```
gustav@sp20:../MPI 10:28:03 !558 $ cat bad_life.c
#include <stdio.h>
#include <mpi.h>

void main(int argc, char *argv[])
{
  int taskid;
  MPI_Status stat;

  /* Find out number of tasks/nodes. */
  MPI_Init (&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &taskid);

  if ( (taskid % 2) == 0 ) {
    char *send_message = NULL;

    send_message = (char *) malloc(1);
    strcpy(send_message, "Forty Two");
    MPI_Send(send_message, 1, MPI_CHAR, taskid+1, 0, MPI_COMM_WORLD);
    free(send_message);
  } else {
    char *recv_message = NULL;

    MPI_Recv(recv_message, 1, MPI_CHAR, taskid-1, 0, MPI_COMM_WORLD, &stat);
    printf("The answer is %s\n", recv_message);
    free(recv_message);
  }

  printf("Task %d complete.\n", taskid);
  MPI_Finalize();
  exit(0);
}
gustav@sp20:../MPI 10:28:06 !559 $
```

Compile and link

```
gustav@sp20:../MPI 10:28:06 !559 $ mpcc -g -o bad_life bad_life.c
gustav@sp20:../MPI 10:29:11 !560 $
```

Run

```
gustav@sp20:../MPI 10:29:11 !560 $ poe bad_life -procs 8
Task 0 complete.
Task 2 complete.
Task 4 complete.
Task 6 complete.
ERROR: 0031-250  task 3: Segmentation fault
ERROR: 0031-250  task 7: Segmentation fault
ERROR: 0031-250  task 2: Terminated
ERROR: 0031-250  task 0: Terminated
ERROR: 0031-250  task 4: Terminated
ERROR: 0031-250  task 6: Terminated
ERROR: 0031-250  task 1: Terminated
ERROR: 0031-250  task 5: Terminated
gustav@sp20:../MPI 10:30:05 !561 $
```

Debug

```
gustav@sp20:../MPI 10:30:05 !561 $ ls -FCsd core*
   0 coredir.3/     0 coredir.7/
gustav@sp20:../MPI 10:30:55 !562 $ ls -FCs coredir.3
total 5
   5 core
gustav@sp20:../MPI 10:31:20 !563 $ dbx bad_life coredir.3/core
Type 'help' for help.
reading symbolic information ...
[using memory image in coredir.3/core]

Segmentation fault in moveeq.memcpy [/usr/lpp/ppe.poe/lib/ip/libmpci.a] at 0xd0b49d9c
0xd0b49d9c (memcpy+0x11c) 7ca01d2a       stsx   r5,r0,r3
(dbx) where
moveeq.memcpy() at 0xd0b49d9c
cpfromdev() at 0xd0b48b40
readdatafrompipe() at 0xd0b46aac
readfrompipe() at 0xd0b4d254
kickpipes() at 0xd0b4833c
mpci_recv() at 0xd0b54298
_mpi_recv(??, ??, ??, ??, ??, ??, ??) at 0xd0a15fc8
MPI__Recv(??, ??, ??, ??, ??, ??, ??) at 0xd0a14888
main(argc = 1, argv = 0x2ff228f0), line 23 in "bad_life.c"
(dbx) func main
(dbx) list 23
   23       MPI_Recv(recv_message, 1, MPI_CHAR, taskid-1, 0, MPI_COMM_WORLD, &stat);
(dbx) print recv_message
"recv_message" is not active
(dbx) print stat
(source = 0, tag = 0, error = 1, val1 = 0, val2 = 0, val3 = -559038737, val4 = -559038737, val5 = -55903873
(dbx) print taskid
3
(dbx) quit
gustav@sp20:../MPI 10:33:54 !564 $
```

Run under the parallel debugger pdbx:

```
gustav@sp20:../MPI 10:38:59 !568 $ pdbx bad_life -procs 8
pdbx Version 2, Release 3 -- Oct 13 1998 21:45:00

  2:reading symbolic information ...
  0:reading symbolic information ...
  1:reading symbolic information ...
  3:reading symbolic information ...
  4:reading symbolic information ...
  5:reading symbolic information ...
  7:reading symbolic information ...
  6:reading symbolic information ...
  0:[1] stopped in main at line 10
  0:   10     MPI_Init (&argc, &argv);
  2:[1] stopped in main at line 10
  2:   10     MPI_Init (&argc, &argv);
  3:[1] stopped in main at line 10
  3:   10     MPI_Init (&argc, &argv);
  4:[1] stopped in main at line 10
  4:   10     MPI_Init (&argc, &argv);
  7:[1] stopped in main at line 10
  7:   10     MPI_Init (&argc, &argv);
  1:[1] stopped in main at line 10
  1:   10     MPI_Init (&argc, &argv);
  5:[1] stopped in main at line 10
  5:   10     MPI_Init (&argc, &argv);
  6:[1] stopped in main at line 10
  6:   10     MPI_Init (&argc, &argv);
0031-504  Partition loaded ...

pdbx(all) cont
  0:Task 0 complete.
  2:Task 2 complete.
  6:Task 6 complete.
  4:Task 4 complete.
  5:
  5:Segmentation fault in @moveeq._moveeq [/usr/lpp/ppe.poe/lib/ip/libmpci.a] at 0xd0b42d9c
  5:0xd0b42d9c (memmove+0x11c) 7ca01d2a       stsx   r5,r0,r3
  1:
  1:Segmentation fault in @moveeq._moveeq [/usr/lpp/ppe.poe/lib/ip/libmpci.a] at 0xd0b49d9c
  1:0xd0b49d9c (memmove+0x11c) 7ca01d2a       stsx   r5,r0,r3
  7:
  7:Segmentation fault in @moveeq._moveeq [/usr/lpp/ppe.poe/lib/ip/libmpci.a] at 0xd06f4d9c
  7:0xd06f4d9c (memmove+0x11c) 7ca01d2a       stsx   r5,r0,r3
  3:
  3:Segmentation fault in @moveeq._moveeq [/usr/lpp/ppe.poe/lib/ip/libmpci.a] at 0xd0b49d9c
  3:0xd0b49d9c (memmove+0x11c) 7ca01d2a       stsx   r5,r0,r3
^C
pdbx-subset(all) on 7

pdbx(7) where
  7:@moveeq.memmove() at 0xd06f4d9c
  7:cpfromdev() at 0xd06f3b40
  7:readdatafrompipe() at 0xd06f1aac
  7:readfrompipe() at 0xd06f8254
  7:kickpipes() at 0xd06f333c
  7:mpci_recv() at 0xd06ff298
  7:_mpi_recv(??, ??, ??, ??, ??, ??, ??) at 0xd0653fc8
  7:MPI__Recv(??, ??, ??, ??, ??, ??, ??) at 0xd0652888
  7:main(argc = 1, argv = 0x2ff228c8), line 23 in "bad_life.c"

pdbx(7) func main
pdbx(7) list 23
  7:   23       MPI_Recv(recv_message, 1, MPI_CHAR, taskid-1, 0, MPI_COMM_WORLD, &stat);

pdbx(7) print recv_message
  7:"recv_message" is not active

pdbx(7) quit
gustav@sp20:../MPI 10:41:43 !569 $
```

8.6.1 Deadlocks

Consider the following program:

```
#include <mpi.h>

#define PIXEL_WIDTH 50
#define PIXEL_HEIGHT 50

int First_Line = 0;
int Last_Line = 0;

void main(int argc, char *argv[])
{
  int numtask;
  int taskid;

  MPI_Init( &argc, &argv);
  MPI_Comm_size( MPI_COMM_WORLD, &numtask );
  MPI_Comm_rank( MPI_COMM_WORLD, &taskid );

  if ( taskid == 0 )
    collect_pixels( taskid, numtask );
  else
    compute_pixels( taskid, numtask );

  printf( "Task %d waiting to complete.\n", taskid );

  MPI_Barrier( MPI_COMM_WORLD );
  printf( "Task %d complete.\n", taskid );
  MPI_Finalize();
  exit();
}

compute_pixels( int taskid, int numtask )
{
  int section;
  int row, col;
  int pixel_data[2];
  MPI_Status stat;

  printf( "Compute #%d: checking in\n", taskid );

  section = PIXEL_HEIGHT / ( numtask - 1 );

  First_Line = ( taskid - 1 ) * section;
  Last_Line = taskid * section;

  for ( row = First_Line; row < Last_Line; row++ )
    for ( col = 0; col < PIXEL_WIDTH; col++ )
      {
        pixel_data[0] = row;
        pixel_data[1] = col;
        MPI_Send( pixel_data, 2, MPI_INT, 0, 0, MPI_COMM_WORLD );
      }

  printf( "Compute #%d: done sending.\n", taskid );
  return;
}

collect_pixels( int taskid, int numtask )
{
  int pixel_data[2];
  MPI_Status stat;
  int mx = PIXEL_HEIGHT * PIXEL_WIDTH;

  printf( "Control #%d: No. of nodes used is %d\n", taskid, numtask );
  printf( "Control: expect to receive %d messages\n", mx );

  while ( mx > 0 )
    {
      MPI_Recv( pixel_data, 2, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
                MPI_COMM_WORLD, &stat );
      mx--;
    }

  printf( "Control node #%d: done receiving.\n", taskid );
  return;
}
```

Compile this program with:

```
gustav@sp20:../MPI_hangs 09:28:22 !505 $ mpcc -g -o rtrace_bug rtrace_bug.c
gustav@sp20:../MPI_hangs 09:28:39 !506 $
```

And run it as follows:

```
<:28:39 !506 $ rtrace_bug -procs 4 -labelio yes -tracelevel 9
  0:Control #0: No. of nodes used is 4
  0:Control: expect to receive 2500 messages
  3:Compute #3: checking in
  2:Compute #2: checking in
  1:Compute #1: checking in
  2:Compute #2: done sending.
  2:Task 2 waiting to complete.
  3:Compute #3: done sending.
  3:Task 3 waiting to complete.
  1:Compute #1: done sending.
  1:Task 1 waiting to complete.
^CERROR: 0031-250  task 3: Interrupt
ERROR: 0031-250  task 0: Interrupt
ERROR: 0031-250  task 1: Interrupt
ERROR: 0031-250  task 2: Interrupt
```

```
gustav@sp20:../MPI_hangs 09:32:05 !507 $
```

This program will hang. When it does so, interrupt it with `^C`. The `-labelio yes` option makes `poe` label output from the parallel tasks by task ids, so that you can clearly see who writes what.

Now invoke `vt` on the trace with the command:

```
gustav@sp20:../MPI_hangs 09:35:07 !508 $ vt -tracefile rtrace_bug.trc &
[1] 8302
gustav@sp20:../MPI_hangs 09:35:47 !509 $
```

If you get complaints about lack of colours, close down the `vt`, then close down `netscape` or any other "colour hog" that may be running on your display, and restart the `vt`.

Use the `vt` to check what happens towards the end of the trace: invoke the `Interprocessor Communication` window and the `Source Code` window. Observe that after some initial messaging activity, the program gets hung on

task 0 `MPI_Recv` in `collect_pixels`

other tasks `MPI_Barrier` in `main`

Use magnifying glass to stretch the horizontal bars in the `Interprocessor Communication` window. Left click on the bars towards the end of the program in order to see the communication operations that the program hangs on.

Identify the cause for the hang and fix the program.

8.6.2 Using the PE Debugger

Restart the same program, `rtrace_bug`, but without the tracefile option this time:

```
gustav@sp20:../MPI_hangs 12:22:06 !518 $ rtrace_bug -procs 4 -labelio yes
  0:Control #0: No. of nodes used is 4
  0:Control: expect to receive 2500 messages
  1:Compute #1: checking in
  3:Compute #3: checking in
  2:Compute #2: checking in
  2:Compute #2: done sending.
  2:Task 2 waiting to complete.
  3:Compute #3: done sending.
  3:Task 3 waiting to complete.
  1:Compute #1: done sending.
  1:Task 1 waiting to complete.
```

When the program hangs, in another window type:

```
<22:06 !504 $ ps -u gustav | grep poe | grep -v grep
    43098 34200  pts/1  0:00 poe
gustav@sp20:../MPI_hangs 13:09:01 !505 $
```

This gives us the POE process id number, which, in this case is 34200 (43098 is my uid number).

Now, in the same window attach the `pedb` debugger to the POE process:

```
gustav@sp20:../MPI_hangs 13:18:50 !507 $ pedb -a 34200
pedb Version 2, Release 3 -- Oct 13 1998 21:56:50
Warning: Cannot convert string "Rom10.500" to type FontStruct
```

A window will pop up listing all four tasks and their PID numbers on respective nodes.

Press `Attach All` button. The original window will go away, and you'll get a very large multi-panelled window filling the whole display. The `Stack` panel shows stack listings for all participating processes. You'll see that they all hang on internal MPI function calls, which do not have line numbers. But as you go down the stack you eventually find function calls with reference to appropriate line numbers within the code, e.g., task 0 should flag:

```
collect_pixels(), line 68
```

whereas the other tasks should flag:

```
main(), line 25
```

Double click on the line `collect_pixels()` in the task 0 stack listing: the code should now appear in the large window on the left with the offending line, in this case

```
MPI_Recv( pixel_data, 2, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, ...
```

Go to the `Global Data` panel (it may be hidden, in which case you will need to stretch it a little so that it will show its window and push buttons) and right click on the `Task 0` push button. A small menu will pop up, select `Show All`. Repeat this for all other tasks.

Look at the local data values. Observe that for task 0 `mx` is 100, which means that task 0 thinks that it is still going to receive 100 messages.

You can look the same way at the other tasks and you'll find that they're all stuck waiting at the barrier.

The problem is therefore solved. Task 0 expected to receive 2500 messages, but received 2400 only.

8.7 Assignment

This is simultaneously a mid-term assignment and the final take home exam. The assignment should exercise you in putting together some of the ideas related to *pushing particles* that we have talked about and give you an opportunity to try your hands on MPI too.

The assignment is due by Friday 14th of May 1999 or earlier if you're so inclined. The assignment results should be placed in your AFS directory, preferably in a separate subdirectory. There should be an accompanying `README` file, which explains the content of other files in that directory. You should notify me by e-mail, when you have submitted your assignment.

This assignment is *for your benefit only*, i.e., it is *not* related to your final grade, that being based on your class attendance, participation in laboratory exercises, questions asked during the course, and so on...

Remember: as this is a PhD level course, you're now colleagues, and not pupils. And one doesn't grade one's colleagues. But, of course, one is perfectly at ease rejecting their papers and grant applications.

So, here's the assignment itself:

Exercise 1 Write an MPI (C or F90) or a High Performance Fortran program which traces trajectories of a swarm of dust particles in a central gravitational field. The force that acts on every dust particle is

$$\boldsymbol{F} = -\frac{Mm}{r^2}\frac{\boldsymbol{r}}{r} \tag{8.1}$$

where $\boldsymbol{r}$ is the *guiding vector* of the dust particle, and the source of the force is located at $\boldsymbol{r} = 0$. The mass associated with the source is M and the mass of the dust particle is m. We have also assumed that the gravitational constant $G = 1$.

Because

$$m\boldsymbol{a} = \boldsymbol{F} \tag{8.2}$$

the mass of the dust particle cancels out and we get:

$$\frac{\mathrm{d}^2\boldsymbol{r}}{\mathrm{d}t^2} = -\frac{M}{r^2}\frac{\boldsymbol{r}}{r} \tag{8.3}$$

For simplicity assume that $M = 1$, so that

$$\frac{\mathrm{d}^2\boldsymbol{r}}{\mathrm{d}t^2} = -\frac{1}{r^2}\frac{\boldsymbol{r}}{r} \tag{8.4}$$

initial condition The potential energy of a dust particle in the gravitational field is given by

$$U(\boldsymbol{r}) = -\frac{Mm}{r} \tag{8.5}$$

Observe that $U(\boldsymbol{r})$ becomes $-\infty$ for $r = 0$ and converges to 0 as $r \to \infty$. Gravitational energy is always negative.

The kinetic energy of the particle is

$$K(\boldsymbol{v}) = \frac{mv^2}{2} \tag{8.6}$$

The kinetic energy of dust particles is always positive.

The total energy of a particle is

$$E(\boldsymbol{r}, \boldsymbol{v}) = \frac{mv^2}{2} - \frac{Mm}{r} \tag{8.7}$$

If the total energy is negative the particle is trapped in the gravitational well.

Generate 10,000 particle positions and velocities at random ensuring that

1. each particle is confined within a sphere of radius 1, where the centre of the sphere coincides with the source of the central force;
2. the energy of each particle is negative.

evolution Evolve the system using either Runge-Kutta method or Bulirsch Stoer method for at least 3 revolutions of the slowest particle around the centre of the system.

measurements For each particle perform the following measurements:

1. measure the total energy of the particle: does the total energy remain constant? It should.
2. measure the angular momentum of the particle:

$$\boldsymbol{L} = m\boldsymbol{r} \times \boldsymbol{v} \tag{8.8}$$

   where $\times$ is the vector product (also called the cross product) of two vectors. $\boldsymbol{L}$ should stay constant both with regards to its length and with regards to its direction. Does it?
3. as the particle moves around the central point of the system it assumes positions $\boldsymbol{r}(t_1), \boldsymbol{r}(t_2), \ldots$. If $t_2 - t_1$ is very small, then the area of the sector whose apex is located at the centre of the system and whose two sides are given by $\boldsymbol{r}(t_1)$ and $\boldsymbol{r}(t_2)$ is

$$\mathcal{A} = |\boldsymbol{r}(t_1) \times \boldsymbol{r}(t_2)| \tag{8.9}$$

   The time $\Delta t = t_2 - t_1$ taken to traverse from $\boldsymbol{r}(t_1)$ to $\boldsymbol{r}(t_2)$ should be proportional to $\mathcal{A}$ along the whole trajectory of an individual dust particle. Is it?
4. measure the time it takes for the particle to travel the full circle around the centre of the system. You may have to resort to interpolation in order to measure that time exactly. Let us call that time T.
5. measure the smallest distance between the particle and the centre of the system attained throughout a single revolution around the centre of the system. You may have to resort to interpolation in order to measure that distance exactly. Let us call that distance r_{min}
6. evaluate T^2/r^3_{min} for all particles. The number should be the same for all of them. Is it?

plots Use GNUplot to plot the following quantities for selected particles:

- the total energy, $\boldsymbol{L}$, and $\mathcal{A}/\Delta t$ in function of time.
- the trajectory, i.e., $\boldsymbol{r}$ in function of time

- T^2/r^3_{min} in function of r_{min}

Exercise 2 This time assume that all particles of the system interact with each other as follows:

$$m_i \frac{\mathrm{d}^2 \boldsymbol{r}_i}{\mathrm{d}t^2} = \sum_{j \neq i} - \frac{m_i m_j}{|\boldsymbol{r}_i - \boldsymbol{r}_j|^2} \frac{\boldsymbol{r}_i - \boldsymbol{r}_j}{|\boldsymbol{r}_i - \boldsymbol{r}_j|} \tag{8.10}$$

Observe that if $\boldsymbol{r}_j = 0$ then the equation of motion is the same as for the central force. The direction of the vector $\boldsymbol{r}_i - \boldsymbol{r}_j$ points from particle j to particle i, so $-(\boldsymbol{r}_i - \boldsymbol{r}_j)$ points from particle i to particle j, therefore particle i is *attracted* to particle j.

initial condition The total energy of the system is the sum of all kinetic and potential energies of individual particles. The kinetic energy of an individual particle is, as before,

$$K_i = \frac{m_i v_i^2}{2} \tag{8.11}$$

The potential energy of an individual particle is

$$U_i = \sum_{j \neq i} - \frac{m_i m_j}{|\boldsymbol{r}_i - \boldsymbol{r}_j|} \tag{8.12}$$

And the total energy is

$$E = \sum_i K_i + U_i \tag{8.13}$$

- generate 10,000 random positions for the particles within the sphere of radius 1.
- Assume all masses to be 1.
- For each particle evaluate its potential energy and generate random velocity so that the abolute value of its kinetic energy is less than the abolute value of its potential energy. This guarantees that the total energy of the ensemble is negative.

evolution evolve the system for a sufficient time to observe significant changes, e.g., some of the particles should traverse across the volume of the system a couple of times

measurements the total energy of the system E and the total angular momentum:

$$\sum_i m_i \boldsymbol{r}_i \times \boldsymbol{v}_i \tag{8.14}$$

should stay constant. Do they?

observations you may notice some particles acquiring a positive total energy at a cost of some other particles *cooling down*. That is how the ensemble evaporates. At every step look for a particle with the lowest total energy and for a particle with the highest total energy.

plots

- Use GNUplot to plot a number of selected trajectories.
- Use GNUplot to plot the lowest and the highest observed total energies in function of time.
- Use GNUplot to plot the total angular momentum and the total energy of the system in function of time.

Exercise 3 Repeat Exercise 2, but this time assume that half of the particles have $m_i = -1$, and the other half $m_i = +1$. Distribute negative and positive values of m at random over the ensemble.

Chapter 9

Message Passing Interface: Advanced Stuff

9.1 Send Modes

blocking does not return until the message data and envelope have been safely stored away so that the sender is free to access and overwrite the send buffer

standard

- unspecified as to buffering
- can be started regardless of the state of the receiver
- unspecified as to locality: it *may* or *may not* complete depending on or regardless of the state of the receiver
- `MPI_Send`

buffered

- can be started regardless of the state of the receiver
- is *local*, i.e., its completion is independent of the state of the receiver
- `MPI_Bsend`

synchronous

- can be started whether or not a matching receive was posted
- is *non-local*: completes *only* if a matching receive is posted
- `MPI_Ssend`

ready

- can be started *only* when the matching receive has been posted – otherwise an *error* is returned.
- completes regardless of whether the message has been fully received.
- `MPI_Rsend`

9.1.1 Semantics of Send Modes

order messages are non-overtaking

```
CALL mpi_comm_rank(comm, rank, ierr)
IF (rank .EQ. 0 ) THEN
   CALL mpi_bsend(buf1, count, MPI_REAL, 1, tag, comm, ierr)
   CALL mpi_bsend(buf1, count, MPI_REAL, 1, tag, comm, ierr)
ELSE  ! rank .EQ. 1
   CALL mpi_recv(buf1, count, MPI_REAL, 0, MPI_ANY_TAG, comm, status, ierr)
   CALL mpi_recv(buf2, count, MPI_REAL, 0, tag, comm, status, ierr)
END IF
```

progress if a pair of matching send and receives have been initiated on two processes, then at least one of these two operations will complete, independently of other actions in the system:

```
CALL mpi_comm_rank(comm, rank, ierr)
IF (rank .EQ. 0) THEN
   CALL mpi_bsend(buf1, count, MPI_REAL, 1, tag1, comm, ierr)
   CALL mpi_ssend(buf2, count, MPI_REAL, 1, tag2, comm, ierr)
ELSE    ! rank .EQ. 1
   CALL mpi_recv(buf1, count, MPI_REAL, 0, tag2, comm, status, ierr)
   CALL mpi_recv(buf2, count, MPI_REAL, 0, tag1, comm, status, ierr)
END IF
```

fairness there is no guarantee of fairness in handling communication

resource limitations any pending communication consumes system resources if they are limited – errors will result if resources are not available. Users can attach their own buffers with

```
MPI_BUFFER_ATTACH( BUFFER, SIZE, IERROR )
```

and detach them with

```
MPI_BUFFER_DETACH( BUFFER_ADDR, SIZE, IERROR )
```

deadlocks

example 1

```
CALL mpi_comm_rank( comm, rank, ierr )
IF ( rank .EQ. 0 ) THEN
   CALL mpi_send(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
   CALL mpi_recv(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
ELSE    ! rank .EQ. 1
   CALL mpi_recv(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
   CALL mpi_send(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
END IF
```

this program will succeed even if no buffer is available. Standard SENDs can be replaced with synchronous ones.

example 2

```
CALL mpi_comm_rank( comm, rank, ierr )
IF ( rank .EQ. 0 ) THEN
   CALL mpi_recv(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
   CALL mpi_send(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
ELSE    ! rank .EQ. 1
   CALL mpi_recv(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
   CALL mpi_send(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
END IF
```

This program will *always* deadlock, for all SEND modes.

example 3

```
CALL mpi_comm_rank( comm, rank, ierr )
IF ( rank .EQ. 0 ) THEN
   CALL mpi_send(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
   CALL mpi_recv(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
ELSE    ! rank .EQ. 1
   CALL mpi_send(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
   CALL mpi_recv(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
END IF
```

This program will succeed only if communication system can buffer at least `count` reals.

9.2 Nonblocking Communications

Non-blocking communications, send and receive, have a number of advantages, e.g., they won't hang a program and they can be interleaved with useful work, e.g., computation. But they are harder to use.

A non-blocking `send` command really means a `send start`. The function call returns as soon as possible, i.e., as soon as other parts of the operating system and/or hardware can take over. But the `send` process itself may not be complete still for a long time. A separate `send complete` call is necessary to verify that the data has been copied out of the send buffer and that the buffer can now be used for another communication.

The same applies to a non-blocking `receive`. The operation is initialised with a `receive start`, and then, if the operating system and hardware allow for that, it continues in the background, until a separate `receive complete` call is issued, whose purpose is to verify that all the data has been received in the receive buffer.

Non-blocking `send starts` can be used with the same four modes as blocking sends, i.e., *standard*, *buffered*, *synchronous*, and *ready*.

Non-blocking sends can be matched with blocking receives and vice versa.

The synopsis for the non-blocking standard send is as follows. In C:

```
int MPI_Isend(void* buf, int count, MPI_Datatype datatype, int dest,
              int tag, MPI_Comm, comm, MPI_Request *request)
```

In Fortran:

```
mpi_isend(buf, count, datatype, dest, tag, comm, request, ierror)
   <type> buf(*)
   integer count, datatype, dest, tag, comm, request, ierror
```

and likewise for `MPI_Ibsend`, `MPI_Issend`, and `MPI_Irsend`.

The non-blocking call, compared to the blocking one, needs one more parameter: `MPI_Request *request`. This is a so called *opaque* object, which identifies communication operations and matches the operation that initiates the communication with the operation that terminates it.

How do you terminate a non-blocking send?

The easiest way is to issue in C

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

or, in Fortran:

```
mpi_wait(request, status, ierror)
   integer request, status(mpi_status_size), ierror
```

This call will make your process hang until the operation identified by the `request` is complete. The call to `MPI_Wait` deallocates the `request` and sets the request handle to `MPI_REQUEST_NULL`.

To follow `MPI_Isend` *immediately* with `MPI_Wait` is the same as to call `MPI_Send`. But splitting the latter into the former lets you do a number of other things between the calls to `MPI_Isend` and `MPI_Wait`.

The outcome of the operation is returned in `status`, which may be queried using `MPI_Test_cancelled`.

The other way to complete a non-blocking send is to issue in C

```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
```

or in Fortran

```
mpi_test(request, flag, status, ierror)
   logical flag
   integer request, status(MPI_STATUS_SIZE), ierror
```

Unlike `MPI_Wait` this call doesn't hang waiting for the communication request to get completed. It returns right away with `flag = true` if the operation is complete and the value of `request` is set to `MPI_REQUEST_NULL`. Otherwise `flag = false` and the value of `request` remains unchanged. Most commonly you are likely to use `MPI_Test` in a loop: checking if the communication has completed, then doing something else, then checking again, and so on.

The returned `status` can be tested with a call to `MPI_Test_cancelled`.

The non-blocking equivalent of `MPI_Recv` is in C:

```
int MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int source,
              int tag, MPI_Comm comm, MPI_Request, *request)
```

or in Fortran:

```
mpi_irecv(buf, count, datatype, source, tag, comm, request, ierror)
   <type> buf(*)
   integer count, datatype, source, tag, comm, request, ierror
```

Observe that there is no slot for `status` in this call. The `status` may not be ready yet. You will have to use either `MPI_Wait` or `MPI_Test` to obtain the `status`.

`MPI_Irecv` followed immediately by `MPI_Wait` is equivalent to `MPI_Recv`. But, as was the case with `MPI_Send`, having split `MPI_Recv` into `MPI_Irecv` and `MPI_Wait` lets you do some other work in between the two, thus masking the communication.

If you are at the receiving end there are additional calls you can issue in order to probe an incoming message without receiving it. The operations are `MPI_Iprobe`, `MPI_Probe`, and `MPI_Cancel`.

The operation `MPI_Iprobe` has the following synopsis in C:

```
int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag,
               MPI_Status *status)
```

and in Fortran:

```
mpi_iprobe(source, tag, comm, flag, status, ierror)
   logical flag
   integer source, tag, comm, status(mpi_status_size), ierror
```

It is a non-blocking operation. It returns `flag = true` if there is a message that can be received and that matches `source`, `tag`, and `comm`. The status associated with the message is returned in `status` and can be further inspected for the length of the message, type of data, etc. If `flag = false` then there is no message, and nothing worth looking at is returned in `status`. The subsequent `MPI_Recv` will receive the message identified by the probe assuming that no other thread has snatched the message in the meantime.

The blocking counterpart of `MPI_Probe` hangs on until a matching message has been found. There is no `flag` argument there. In C:

```
int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)
```

and in Fortran:

```
mpi_probe(source, tag, comm, status, ierror)
   integer source, tag, comm, status(MPI_STATUS_SIZE), ierror
```

A non-blocking send or a receive can be cancelled, i.e., discarded with, in C:

```
int MPI_Cancel(MPI_Request *request)
```

or, in Fortran:

```
mpi_cancel(request, ierror)
   integer request, ierror
```

This call *marks* the request for cancellation, but the communication itself still doesn't complete, until you issue, in C:

```
int MPI_Request_free(MPI_Request *request)
```

or, in Fortran:

```
mpi_request_free(request, ierror)
   integer request, ierror
```

You can also complete the communication with `MPI_Wait` or `MPI_Test`, but since you're not interested in the message there is little point waiting or testing for it.

Now, how do you examine `status` once you have it?

Status is a structure in C and an array in Fortran with multiple fields. Three fields compulsory and self-explanatory: `status.MPI_SOURCE`, `status.MPI_TAG`, and `status.MPI_ERROR`. In Fortran these are: `status(MPI_SOURCE)`, `status(MPI_TAG)`, and `status(MPI_ERROR)`. But `status` contains also additional information, which is not directly accessible. There is a function call `MPI_Get_count`, which you can use to inquire about the length of the message, for example, before you're going to receive it.

The synopsis of `MPI_Get_count` is, in C:

```
int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)
```

and in Fortran

```
mpi_get_count(status, datatype, count, ierror)
   integer status(MPI_STATUS_SIZE), datatype, count, ierror
```

Here is an example code, which illustrates the use of some of the non-blocking communication functions discussed so far.

```
#include <stdio.h>
#include <mpi.h>

main(argc, argv)
int argc;
char *argv[];
{

  int pool_size, my_rank;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &pool_size);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

  if (my_rank == 0) {

    char send_buffer[BUFSIZ], my_cpu_name[BUFSIZ];
    int my_name_length;
    MPI_Request request;
    MPI_Status status;

    MPI_Get_processor_name(my_cpu_name, &my_name_length);
    sprintf (send_buffer, "Dear Task 1,\n\
Please do not send any more messages.\n\
Please send money instead.\n\
\tYours faithfully,\n\
\tTask 0\n\
\tRunning on %s\n", my_cpu_name);
    MPI_Isend (send_buffer, strlen(send_buffer) + 1, MPI_CHAR,
               1, 77, MPI_COMM_WORLD, &request);
    printf("hello there user, I've just started this send\n\
and I'm having a good time relaxing.\n");
    MPI_Wait (&request, &status);
    printf("hello there user, it looks like the message has been sent.\n");

    if (request == MPI_REQUEST_NULL) {
      printf("\tthe send request is MPI_REQUEST_NULL now\n");
    } else {
      printf("\tthe send request still lingers\n");
    }

  }
  else if (my_rank == 1) {

    char recv_buffer[BUFSIZ], my_cpu_name[BUFSIZ];
    int my_name_length, count;
    MPI_Request request;
    MPI_Status status;

    MPI_Get_processor_name(my_cpu_name, &my_name_length);
    MPI_Irecv (recv_buffer, BUFSIZ, MPI_CHAR, 0, 77, MPI_COMM_WORLD,
               &request);
    printf("hello there user, I've just started this receive\n\
on %s, and I'm having a good time relaxing.\n", my_cpu_name);
    MPI_Wait (&request, &status);
    MPI_Get_count (&status, MPI_CHAR, &count);
    printf("hello there user, it looks like %d characters \
have just arrived:\n", count );
    printf("%s", recv_buffer);

    if (request == MPI_REQUEST_NULL) {
      printf("\tthe receive request is MPI_REQUEST_NULL now\n");
    } else {
      printf("\tthe receive request still lingers\n");
    }

  }
  MPI_Finalize();

}
```

And here is how this program compiles and runs:

```
gustav@sp20:../MPI 18:35:56 !553 $ mpcc -g -o non-block non-block.c
gustav@sp20:../MPI 18:36:45 !554 $ non-block -procs 2 -labelio yes
  1:hello there user, I've just started this receive
  1:on sp19.ucs.indiana.edu, and I'm having a good time relaxing.
  0:hello there user, I've just started this send
  0:and I'm having a good time relaxing.
  0:hello there user, it looks like the message has been sent.
  0:    the send request is MPI_REQUEST_NULL now
  1:hello there user, it looks like 139 characters have just arrived:
  1:Dear Task 1,
  1:Please do not send any more messages.
  1:Please send money instead.
  1:    Yours faithfully,
  1:    Task 0
  1:    Running on sp17.ucs.indiana.edu
  1:    the receive request is MPI_REQUEST_NULL now
gustav@sp20:../MPI 18:36:51 !555 $
```

9.2.1 Exercises

1. Replace non-blocking receive with a blocking probe. Interrogate `status` to find out the size of the message. Allocate dynamically appropriate amount of space to receive the message.

2. Replace `MPI_Wait` in task 1 with a looping `MPI_Test`.

3. Write a short program for 8 tasks, which passes an arbitrary message around using non-blocking sends and receives combined with `MPI_Wait`. Make every task write the message on standard output immediately after having started a send to the next task, but before the completion of the send. Generate a trace file and use the visualization tool, `vt`, to observe the progress of the program. Observe that the communication effectively serializes the program.

9.2.2 Multiple Completions

There are 6 function calls that let a programmer wait for the completion of *any*, *some*, or *all* pending communication operations. These are, in C:

```
int MPI_Waitany(int count, MPI_Request *array_of_requests, int *index,
                MPI_Status *status)
```

and in Fortran:

```
mpi_waitany(count, array_of_requests, index, status, ierror)
   integer count, array_of_requests(*), index, status(MPI_STATUS_SIZE), &
           ierror
```

Similarly we have, in C:

```
int MPI_Testany(int count, MPI_Request *array_of_requests, int *index,
                int *flag, MPI_Status *status)
```

and in Fortran:

```
mpi_testany(count, array_of_requests, index, flag, status, ierror)
   logical flag
   integer count, array_of_requests(*), index, status(MPI_STATUS_SIZE),
           ierror
```

Then we have:

```
int MPI_Waitall(int count, MPI_Request *array_of_requests,
                MPI_Status *array_of_statuses)
```

and in Fortran:

```
mpi_waitall(count, array_of_requests, array_of_statuses, ierror)
   integer count, array_of_requests(*)
   integer array_of_statuses(MPI_STATUS_SIZE, *), ierror
```

This is accompanied by the corresponding `MPI_TESTALL` calls, in C:

```
int MPI_Testall(int count, MPI_Request *array_of_requests, int *flag,
                MPI_Status *arrray_of_statuses)
```

and in Fortran:

```
mpi_testall(count, array_of_requests, flag, array_of_statuses, ierror)
   logical flag
   integer count, array_of_requests(*)
   integer array_of_statuses(MPI_STATUS_SIZE, *), ierror
```

Last, there are the `MPI_WAITSOME` calls. In C:

```
int MPI_Waitsome(int incount, MPI_Request *array_of_requests,
                 int *outcount, int *array_of_indices,
                 MPI_Status *array_of_statuses)
```

and in Fortran:

```
int mpi_waitsome(incount, array_of_requests, outcount, array_of_indices,
                 array_of_statuses, ierror)
   integer incount, array_of_requests(*), outcount, array_of_indices(*)
   integer array_of_statuses(MPI_STATUS_SIZE, *), ierror
```

9.3 Derived Data Types

This stuff is used for polymorphic messages, i.e., messages that are made of compound items comprising various generic data types, e.g., integers, floats, and characters – all in a single data block. Derived data types are also used to transfer data from non-contiguous buffers, e.g., a portion of a matrix. To this effect MPI lets programmers specify mixed and non-contiguous communication buffers, so that objects of various shapes and sizes can be transferred directly without copying.

However, MPI as it is defined and implemented today, has no means of figuring out on its own how such data is laid out in the host language, i.e., C, F77, or F95. This information could, in principle, be obtained by decoding definitions from a symbol table, but no attempt has been made in the MPI definition to incorporate this. Consequently it befalls the programmer to figure this out and then define matching derived MPI datatypes.

Neither does MPI specify how such transfers are to be implemented. It is still possible that an actual MPI implementation would copy all the data to an auxiliary contiguous buffer before transfer. But MPI *semantics* allow for a direct transfer from non-contiguous memory locations, and for a direct transfer of polymorphic data blocks.

An MPI programmer defines a new derived data type by specifying its map and its signature. This can be done recursively, i.e., an already defined data type can be used in constructing a new map. Furthermore, all MPI data type definitions occur during program execution. Consequently various MPI derived types can be associated with the same name as the program unfolds.

A *type map* is a sequence of the following form

```
Typemap = {(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})}
```

In turn a *type signature* is the following sequence

```
Typesig = {type_0, ..., type_{n-1}}
```

General MPI datatypes can be used in all send and receive operations. Basic data types can be thought of as special cases of general data types, for example:

```
MPI_INT = {(int, 0)}
```

The *extent* of a general datatype is a *span* from the first to the last byte occupied by entries in the datatype *rounded* up to satisfy alignment requirements, if any

```
extent(typemap) = ub(typemap) - lb(typemap)
```

where ub stands for the *upper bound* and lb stands for the *lower bound*, for example:

```
extent({(double, 0), (char, 8)}) = 16
```

Here the size is 16 B (16 bytes), rather than 9 B because the data item must be padded to the next word boundary.

In the following we are going to look first at MPI derived data type constructors, then at auxiliary functions that return an address of a variable and an extent, upper and lower bound markers for a defined datatype. Then we'll talk about *committing* and *freeing* a datatype and what it means and finally we'll have a look at some examples.

9.3.1 Datatype Constructors

Contiguous

The simplest constructor is MPI_Type_contiguous, the C language interface of which is given by:

```
MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)
```

In Fortran this function has the following interface:

```
mpi_type_contiguous(count, oldtype, newtype, ierror)
integer count, oldtype, newtype, ierror
```

This function specifies a new derived MPI type, the reference to which is going to be stored on newtype, which comprises count items of type oldtype stored contiguously in the processor's memory. MPI semantics assume that the concatenation preserves the *extent* of the original data.

For example gluing together three objects of type:

```
{(double, 0), (char, 8)}
```

will produce

```
{(double, 0), (char, 8), (double, 16), (char, 24), (double, 32), (char, 40)}
```

If we have used some other function to define the {(double, 0), (char, 8)} and called it, say, named_double, then the new data type could be called three_named_doubles and it's creation in C would be accouplished as follows:

```
MPI_Type_contiguous(3, named_double, &three_named_doubles);
```

We have already encountered this function in our program which calculated forces between particles. There we used function mpi_type_contiguous in order to construct an MPI data type that could be used to transfer a structure comprising 3 coordinates of a particle and its mass between MPI processes.

So how to define an MPI datatype that stores a double precision floating point number and a character? For this we will have to use function MPI_Type_struct. This function has a very complicated synopsis because it lets an MPI programmer construct an object that picks up data items of any types, even derived types, from any memory locations. So we are going to leave the discussion of this function for the time being, instead looking at simpler functions and gradually getting closer to understanding how MPI_Type_struct works.

Vector

The other MPI type constructor we have already seen used is MPI_Type_vector. We used this constructor in the program that exchanged columns of a matrix between processes using MPI_Sendrecv. In C columns are not laid out contiguously (rows are), so we had to call MPI_Type_vector in order to tell the program how to pick data from the matrix so that we would end up transferring the whole column in one go.

In Fortran we would have to do this for matrix rows, because in Fortran columns are laid contiguously and rows are not.

The synopsis of this function in C is:

```
MPI_Type_vector(int count, int blocklength, int stride,
                MPI_Datatype oldtype, MPI_Datatype *newtype)
```

and in Fortran:

```
mpi_type_vector(count, blocklength, stride, oldtype, newtype, ierror)
integer count, blocklength, stride, oldtype, newtype, ierror
```

The function picks up count blocks of data of type MPI_Datatype. Each block is blocklength data items long. The separation between the beginning of one block and the beginning of the next one is stride. The newly constructed data type is now associated with the memory location pointed to by newtype, which has been structured to store all information about this datatype. Within each block data items of type `oldtype` are laid out contiguously.

For example, if the old type is:

```
oldtype = {(double, 0), (char, 8)}
```

then the call to

```
MPI_Type_vector(2, 3, 4, named_double, &six_named_doubles)
```

will create a new MPI data type, which is going to have the following map:

```
newtype =
{(double,  0), (char,  8), (double, 16), (char, 24), (double, 32), (char,  40),
 (double, 64), (char, 72), (double, 80), (char, 88), (double, 96), (char, 104)}
```

In plain language: we are taking two blocks of data. Each block comprises 3 structures of type `named_double` (the map of which is `{(double, 0), (char, 8)}`) concatenated contiguously. The stride is set to the *extent* of 4 objects of the same type, i.e., since the extent of `named_double` is 16, the stride is 64 bytes because $4 \times 16 = 64$.

Function `MPI_Type_contiguous` can be thought of as a special case of `MPI_Type_vector`:

```
MPI_Type_contiguous(count, oldtype, &newtype)
   = MPI_Type_vector(count, 1, 1, oldtype, &newtype)
```

This means that if you ever have to write your own MPI, you can begin by definining `MPI_Type_vector` and then write `MPI_Type_contiguous` as a simple wrapper around the former. But then it may be also the case that you can capitalize on some hardware features and write a faster implementation of `MPI_Type_contiguous` directly.

In `MPI_Type_vector` the stride is defined in terms of an extent of the basic data type used in the operation. There is a special variant of this function that lets you define stride simply in bytes, if you know what they are. This function is called `MPI_Type_hvector` and its synopsis in C is:

```
MPI_Type_hvector(int count, int blocklength, MPI_Aint stride,
                 MPI_Datatype oldtype, MPI_Datatype *newtype)
```

The Fortran synopsis of this function is:

```
mpi_type_hvector(count, blocklength, stride, oldtype, newtype, ierror)
integer count, blocklength, stride, oldtype, newtype, ierror
```

In term of `MPI_Type_hvector` the previous example

```
MPI_Type_vector(2, 3, 4, named_double, &six_named_doubles)
```

would be written as follows:

```
MPI_Type_hvector(2, 3, 64, named_double, &six_named_doubles)
```

Indexed

The *indexed* constructor is a new one. But it works similarly to those collective communication operations that allow for contributions of various sizes from participating processes, e.g., `MPI_Gatherv` or `MPI_Allgatherv`. The `MPI_Type_indexed` synopsis in C is:

```
MPI_Type_indexed(int count, int *array_of_blocklengths,
                 int *array_of_displacements, MPI_Datatype oldtype,
                 MPI_Datatype *newtype)
```

and its Fortran synopsis is:

```
mpi_type_indexed(count, array_of_blocklengths, arrray_of_displacements,
                 oldtype, newtype, ierror)
integer count, array_of_blocklengths(*), array_of_displacements(*),
        oldtype, newtype, ierror
```

This function works as follows. We are going to put together `count` blocks of data of type `oldtype`. The blocks may be of different length now, and because they may be of different lengths their corresponding displacements may be different for every block too, so we can no longer just give a single `stride` value for the blocks. For every block we have to define separately its length and its offset from the beginning of the array, i.e., its displacement. If we are going to have `count` blocks, their lengths and displacements are specified on two arrays of integers, each `count` elements long. The arrays are `array_of_blocklengths` and `array_of_displacements`. But other than that these functions still work much like `MPI_Type_vector`. In fact `MPI_Type_vector` can be viewed as a specific case of `MPI_Type_indexed` and it can be implemented as such. In particular all blocks comprise items of the same data type.

Here is an example that illustrates how this function works. Consider the same `oldtype` as above, i.e., the `named_double`. Its map is:

```
named_double = {(double,0), (char, 8)}
```

We are now going to call:

```
MPI_Type_indexed(2, (3,1), (4,0), named_double, &newtype)
```

What is going to be the `newtype` map? We are going to put together 2 blocks of data items of type `named_double`. The first block has length 3 and offset 4. The second block has length 1 and offset 0. Observe that we are changing here the order of data, sic! Here is the map of this new MPI data type:

```
newtype =
{(double, 64), (char, 72), (double, 80), (char, 88), (double, 96), (char, 104),
 (double,  0), (char,  8)}
```

There is also a variant of this function that lets MPI programmers specify displacements in bytes rather than in *extents* of the basic data type used in the operation. The synopsis of this function in C is:

```
MPI_Type_hindexed(int count, int *array_of_blocklengths,
                  MPI_Aint *array_of_displacements, MPI_Datatype oldtype,
                  MPI_Datatype *newtype)
```

where `array_of_displacements` contains displacements in *bytes*. The Fortran interface is:

```
mpi_type_hindexed(count, array_of_blocklengths, array_of_displacements,
                  oldtype, newtype, ierror)
integer count, array_of_blocklengths(*), array_of_displacements(*)
        oldtype, newtype, ierror
```

Structures

The last, and the most complex MPI data type constructor is `MPI_Type_struct`. This function lets MPI programmers do all that they can do with `MPI_Type_indexed`, *and* on top of all that the items picked up from various memory locations can have different types. The items, as before, are assumed to be present in contiguous blocks at every location pointed to by the `displacement` array and within those blocks they are all assumed to be of *the same* type. But the types may vary from block to block. Here is the synopsis of this function in C:

```
MPI_Type_struct(int count, int *array_of_blocklengths,
                MPI_Aint *array_of_displacements, MPI_Datatype *array_of_types,
                MPI_Datatype *newtype)
```

and in Fortran:

```
mpi_type_struct(count, array_of_blocklengths, array_of_displacements,
                array_of_types, newtype, ierror)
```

In summary the function works as follows: we are going to collect `count` blocks of data. Each block comprises a number of elements given by the corresponding entry in `array_of_blocklengths`. Each block begins at a location given by the corresponding entry in `array_of_displacements`. The type of data in each block is given by the corresponding entry in `array_of_types`. What are the displacements measured in? This is not a trivial question, because now every block can comprise elements of a different type. There is only one way to measure the displacements in this context. They have to be measured in bytes. Consequently there is no `MPI_Type_hstruct` function. You may say that `MPI_Type_struct` *is* the `MPI_Type_hstruct`: the displacements are measured in bytes and there is no no-byte version of this function.

Here is an example of how this function works. Let

```
type1 = {(double, 0), (char, 8)}
```

The call:

```
MPI_Type_struct(3, (2, 1, 3), (0, 16, 26), (MPI_FLOAT, type1, MPI_CHAR),
                &newtype)
```

constructs a new data type with the following map:

```
newtype =
{(float, 0), (float, 4), (double, 16), (char, 24), (char, 26), (char, 27),
 (char, 28)}
```

The call grabs 3 data blocks from locations given by the following displacements in bytes: 0, 16, and 26. The first block comprises 2 floating point numbers. The second block comprises one item of type `type1`, which is a double precision number followed by a character. The third block comprises 3 characters. Observe that the characters of the third block commence from a half-word boundary, i.e., byte number 26, and then the following 1-byte characters are written one after another without gaps in between. Strings are usually stored like that. It would be tremendously wasteful to pad every 8-bit character with 3 empty bytes (so as to reach a 32-bit boundary on a 32-bit system). Likewise, some architectures allow to pack up to two separate items into one word, so that if you have to pad you only pad up to the half-word boundary.

All these things are obviously very system dependent. If you ever decide to write programs like that you must very carefully annotate what you do. A program that makes 32-bit architecture assumptions may not work on a 64-bit or a 128-bit system. Today 128-bit systems become increasingly common. The new Macintoshes and the new games machines are 128-bit architectures. The IBM Power-3 and Power-4 chips are in a way also 128-bit chips (split into 2×64 for easier coding and better register utilization). As these systems become more popular people who wrote MPI programs for 32-bit or 64-bit architectures utilizing hand-defined derived MPI data types are going to get into trouble. Low level coding is always very difficult to port.

9.3.2 Data Type Inquiry Functions

To a limited extend MPI assists programmers in writing portable programs, even programs that construct derived data types "manually" by providing a range of inquiry functions which can be used to ask about the address of a given location, extent of a given derived data type, its size (this is not the same as an extent), and lower and upper bound markers for a given derived data type. This way a program can be constructed that checks for these things and then compare them with assumptions made in the program. If any inconsistencies are found a program may issue a message and abort, or take a corrective action if possible. These functions can be also used to contruct data types dynamically, thus making the whole procedure more portable.

The first one of these functions is a function that returns an address of an object in... *bytes*. The bytes are counted from the beginning of the usable memory. There is an MPI constant called MPI_BOTTOM, which marks the beginning of memory. Once you have absolute addresses of various objects, as returned by function MPI_address and MPI_BOTTOM you can give displacements to any of the constructor functions in terms of those addresses – and, the pointer to the buffer, of course, must then be given in terms of MPI_BOTTOM.

The synopsis of function MPI_Address in C is:

```
MPI_Address(void* location, MPI_Aint *address)
```

and in Fortran:

```
mpi_address(location, address, ierror)
<type> location(*)
integer address, ierror
```

Observe that the address, as returned by a call to `MPI_Address` is not the same as a pointer. A pointer in C, or in Fortran, or in Pascal *contains* an address, but it may also contain information about the type of the variable that the pointer points to. Here we are concerned with addresses only, and we want them given to us in bytes.

The following example code shows how you can use `MPI_Address` to find a distance, in bytes, between the beginning of an array and a particular location in that array. You can then use this information in order to write a more portable program, which would work the same regardless of how much space is taken by a real number (it may be 32-bits or 64-bits depending on the architecture of a machine).

```
real A(100, 100)
integer i1, i2, diff

call mpi_address(a(1,1), i1, ierror)
call mpi_address(a(10,10), i2, ierror)
diff = i2 - i1
```

The next function can be used to find the *extent* of a derived data type. The function is `MPI_Type_extent`. The extent is also given in bytes. The C language synopsis of this function is:

```
MPI_Type_extent(MPI_Datatype datatype, MPI_Aint *extent)
```

and its Fortran interface is:

```
mpi_type_extent(datatype, extent, ierror)
integer datatype, extent, ierror
```

The *extent* is not the same as the *size* of a given data-type. The extent is a difference between the upper bound marker and the lower bound marker for a given data type. There may be a lot of empty space in between, some padding, etc. All this is included in the *extent*.

On the other hand the size of a data type is the number of bytes that are going to be transmitted when a process `MPI_Send`s a message that contains this data type. In this case the padding and the empty space that separate and wrap various components of the data type are not going to be transmitted. Instead MPI is going to take all the real data from its various locations pointed to by the `displacements`, `lengths`, and `types` arrays and send just that. The synopsis of the function that returns the *size* of a derived data type is:

```
MPI_Type_size(MPI_Datatype datatype, int *size)
```

in C and

```
mpi_type_size(datatype, size, ierror)
integer datatype, size, ierror
```

in Fortran.

The lower and the upper bounds of any MPI data type, derived or not, can be obtained by calling functions `MPI_Type_lb` and `MPI_Type_ub`. Both have a similar synopsis, e.g.,

```
int MPI_Type_lb(MPI_Datatype datatype, MPI_Aint* displacement)
```

in C and

```
mpi_type_ub(datatype, displacement, ierror)
integer datatype, displacement, ierror
```

in Fortran.

But you can also set the lower and upper bound manually on a newly defined MPI typed. For example, you may wish to pad the type on both sides. Then you would not rely on the system generating the upper and the lower bound, but, instead, you would define your own bounds. This can be done by calling `MPI_Type_struct` and using predefined pseudo-datatypes `MPI_LB` and `MPI_UB` to mark the bounds.

9.3.3 Commit and free

We have already enountered functions `MPI_Type_commit` and `MPI_Type_free` so here I just give their definitions for completeness:

- C:

  ```
  MPI_Type_commit(MPI_Datatype *datatype)
  ```

  Fortran:

  ```
  mpi_type_commit(datatype, ierror)
     integer datatype, ierror
  ```

In principle this function is supposed to generate a formal description of the communication buffer, which is then going to be used by consequent communication operations on this or other similar buffers. The system, on receiving this call, may *compile* the datatype and generate an internal representation of it. The most convenient transfer mechanism, for this datatype, may be selected too.

- C:

  ```
  MPI_Type_free(MPI_Datatype *datatype)
  ```

  Fortran:

  ```
  mpi_type_free(datatype, ierror)
     integer datatype, ierror
  ```

  The `datatype` is set to `MPI_DATATYPE_NULL`. All pending communications that use this datatype complete normally. Datatypes derived from the freed datatype are not affected.

9.3.4 Counting the elements

The function

```
MPI_Get_elements(MPI_Status *status, MPI_Datatype datatype, int *count)
```

returns the number of *basic* elements of `datatype` received in a message. There is a subtle difference between this function and `MPI_Get_count`, and it shows in the situations in which `datatype` is used to define a complex layout of data in the receiver memory and does not represent a basic unit of data for transfers.

Example:

```
...
call mpi_type_contiguous(2, MPI_REAL, type2, ierr)
call mpi_type_commit(type2, ierr)
...
call mpi_comm_rank(comm, rank, ierr)
if(rank .eq. 0) then
   call mpi_send(a, 2, mpi_real, 1, 0, comm, ierr)
   call mpi_send(a, 3, mpi_real, 1, 0, comm, ierr)
else
   call mpi_recv(a, 2, type2, 0, 0, comm, stat, ierr)
   call mpi_get_count(stat, type2, i, ierr)    ! returns i = 1
   call mpi_get_elements(stat, type2, i, ierr) ! returns i = 2
   call mpi_recv(a, 2, type2, 0, 0, comm, stat, ierr)
   call mpi_get_count(stat, type2, i, ierr)    ! returns i = MPI_UNDEFINED
   call mpi_get_elements(stat, type2, i, ierr) ! returns i = 3
end if
```

9.3.5 Particles Again

In this section we are going to have a yet another look at the problem of handling particles in MPI. But our particles are now going to be described as God meant them to be, i.e., they are going to have some character descriptors, some class descriptors (this is going to be an integer), and some floating point descriptors, e.g., a vector of coordinates and velocities. And the we are going to do various things with them in order to illustrate the use of the functions discussed in this section.

The following is not a real, complete code. Instead these are various excerpts from a code that you may wish to write one day, so the ideas presented here will come handy.

So let us assume that we have done our `#include` and `main` and now we get down to the interesting bits. First some declarations. Let a structure that represents particles be defined as follows:

```
struct Partstruct
   {
   int    class;  /* particle class */
   double d[6];   /* particle coordinates */
   char   b[7];   /* some additional information */
   };
```

Now we define an array that is going to house 1000 of these particles:

```
struct Partstruct    particle[1000];
```

In order to represent the particles to MPI we have to prepare the ground for calling `MPI_Struct`. So we begin from the following definitions:

```
MPI_Datatype Particletype;
MPI_Datatype type[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};
int          blocklen[3] = {1, 6, 7};
MPI_Aint     disp[3];
int          base;
```

Given a particle structure, as implemented by the compiler, and we really don't know a priori how that is going to be done, and we want to write a portable program, we obtain the information about data layout within the structure `Partstruct` by interrogating this structure with function `MPI_Address`:

```
MPI_Address( particle, disp);
MPI_Address( particle[0].d, disp+1);
MPI_Address( particle[0].b, disp+2);
base = disp[0];
for (i=0; i <3; i++) disp[i] -= base;
```

We have subtracted `disp[0]` from all other `disps` so as to obtain displacements from the beginning of a buffer, as opposed to absolute addresses. This way we don't have to assume while writing the program that data is going to be laid out in some specific way.

Now we are ready to construct an MPI structure that matches the C-defined structure:

```
MPI_Type_struct( 3, blocklen, disp, type, &Particletype);
```

Looking at the definitions of the variables used in this call we are constructing a polymorphic data type which is made of 3 blocks of data. The first block contains 1 variable of type `MPI_INT`, and its relative displacement is 0. The second block contains 6 variables of type `MPI_DOUBLE` and its displacement is given by `disp[1]`, which we have calculated by looking at the absolute address of this block for a real object of type `Partstruct`. The third block of data comprises 7 characters, and its displacement is given by `disp[2]`.

Now we have to *commit* this newly created MPI data type and from this point onwards we can use it in order to send and receive particles:

```
MPI_Type_commit( &Particletype);
MPI_Send( particle, 1000, Particletype, dest, tag, comm);
```

But we can do better. Say that some of the particles happen to be of *class* zero and that we want to send *only* particles in this class. How to do this?

Let us begin by adding some more definitions for a new data type:

```
MPI_Datatype Zparticles;
MPI_Datatype Ztype;
MPI_Aint     zdisp[1000];
int zblock[1000], j, k;
int zzblock[2] = {1,1};
MPI_Aint     zzdisp[2];
MPI_Datatype zztype[2];
```

When we get through to the part of the code that is going to construct the new data type that corresponds to particles of class zero we begin by computing displacements for these particles:

```
j = 0;
for(i=0; i < 1000; i++)
  if (particle[i].class==0)
     {
     zdisp[j] = i;
     zblock[j] = 1;
     j++;
     }
```

Since we have already defined an MPI type for a particle we can use a simpler function, namely `MPI_Type_indexed` in order to define the type for particles of class zero:

```
MPI_Type_indexed( j, zblock, zdisp, Particletype, &Zparticles);
```

This data type comprises `j` blocks, each of which contains one particle. The blocks are picked up from locations given by the array `zdisp`. The newly constructured data type is written on `Zparticles`.

We could now send the lot across to another process in just one operation. But it would be nice to also add information about the number of particles we are going to send across. This number sits inside `j`, and we are going to prepend it to `Zparticles` and create another compound MPI data type, which comprises one integer, namely the number of particles that we are going to send, and then all those particles.

First we find what are absolute addresses of `j` and of the first `particle`, then we write those addresses on `zzdisp`, which are the displacements for the new data type. The array that specifies the corresponding types

```
MPI_Address(&j, zzdisp);
MPI_Address(particle, zzdisp+1);
```

Before we can call `MPI_Type_struct` we also have to construct an array of types:

```
zztype[0] = MPI_INT;
zztype[1] = Zparticles;
```

Finally we construct this new type that comprises an integer followed by all particles of class zero, we commit this type (observe that we didn't have to commit the intermediate type `Zparticles`) and send the lot across. Because we have used *absolute* addresses, the pointer to the buffer is `MPI_BOTTOM`.

```
MPI_Type_struct(2, zzblock, zzdisp, zztype, &Ztype);
MPI_Type_commit( &Ztype);
MPI_Send( MPI_BOTTOM, 1, Ztype, dest, tag, comm);
```

Now let us say that we want to send the first two coordinates for all particles, i.e., `particle[j].d[0]` and `particle[j].d[1]`, for all `j`. Here are the definitions that we have to introduce to accomplish this task:

```
MPI_Datatype Allpairs;
MPI_Aint sizeofentry;
```

Since we have already constructed the MPI type for a single particle, that type is `Particletype`, we can use this in order to find stride in bytes:

```
MPI_Type_extent( Particletype, &sizeofentry);
```

The type `Allpairs` can be now defined by calling `MPI_Type_hvector`:

```
MPI_Type_hvector( 1000, 2, sizeofentry, MPI_DOUBLE, &Allpairs);
```

Here we are taking 1000 blocks of 2 double precision floating point numbers each from an otherwise unspecified array of data (it may be polymorphic) and the blocks are separated by a stride of `sizeofentry`. We commit this newly constructed MPI data type, then locate the *first* coordinate of the *first* particle and send the whole lot in one go:

```
MPI_Type_commit( &Allpairs);
MPI_Send( particle[0].d, 1, Allpairs, dest, tag, comm);
```

9.3.6 Pack and unpack

MPI provides PVM-like calls `MPI_Pack` and `MPI_Unpack`. These can be used to pack data from a data type with a complex layout into a contiguous buffer before sending. On receiving, the data can be *unpacked* and placed again into a new complex layout.

These two functions are provided largely for backward compatibility with older PVM codes. But they can be also used for more elaborate situations, for example, when a message is received in several chunks and the way that later chunks are going to be interpreted may depend on what's in the earlier ones.

Because the memory lay-out of the data type from which data is plucked for sending needs to be known in advance packing is unlikely to be of much use in transmitting lists, trees, and graphs, unless the latter live within the user-defined arrays and appropriate maps are available.

So how would you send a tree or a graph across? You would have to traverse it, extract all the data, copy it onto a contiguous buffer, provide a prescription for reconstruction of the tree on the other side, and then transmit both types of the information.

But sometimes people implement trees and linked lists themselves within finite size arrays, e.g., arrays of pairs or arrays of triples – if this is the case, then the whole array can be transmitted remembering, of course, about defining array entries for MPI as appropriate datatypes.

9.3.7 Exercises

Write a simple program, which builds a tree on an array of the following structures:

```
    {char, int, int}
```

The first `int` points to the location, i.e., an index value within the array, of the left branch and the second `int` points to the location, i.e., an index value within the array, of the right branch. The value stored at the node of the tree is going to be a character. A negative value, e.g., -1 stored in any of the `int` locations means `NULL`. Zero means the root of the tree.

Define the whole array as a single data type, i.e., a tree. Find the size and the extent of the tree.

Generate dynamically a new MPI type that corresponds to one of the branches of the tree, and send it across to a slave process.

Make both the master process and the slave process print the branch on stdout.

Are they identical?

9.4 Collective Communication

9.4.1 Broadcast, Gather, Scatter

9.4.2 Gather-to-all, All-to-All Scatter/Gather

9.4.3 Reductions

9.4.4 User-Defined Reduction Operations

9.4.5 Scan

9.5 Groups, Communicators, and Inter-Communicators

9.6 Process Topologies

9.6.1 Cartesian Constructor

9.6.2 Graph Constructor

9.6.3 Topology Inquiry Functions

9.7 MPI IO

The purpose of MPI/IO is to provide high performance, portable, parallel I/O interface to high performance, portable, parallel MPI programs. Parallel I/O is not a daily bread. Although some supercomputer systems in the past offered parallel disk subsystems, e.g., the Connection Machine CM5 had a Scalable Disk Array, SDA, the Connection Machine CM2 had the Data Vault and IBM SP had PIOFS and today it has GPFS, communication with those peripherals was architecture and operating system dependent.

Yet I/O is such an important part of scientific computation that a system that provides parallel CPUs but only sequential I/O can hardly be called a supercomputer.

MPI I/O, which was contributed to the MPI-2 standard by NASA, builds on MPI derived data types and collective communications, so that the resulting semantics are very similar to the former two. MPI/IO files are shared, i.e., multiple processes running on multiple CPUs can operate on a single MPI/IO file all at the same time. The file may be spread over the disk systems that belong to those CPUs or it may be spread over some other parallel disk system which can be accessed by the CPUs over parallel communication channels.

Each MPI file is written as a sequence of *etypes*. An etype, which stands for *elementary datatype* is the unit of data access and positioning. But *etypes* don't really have to be all that elementary. Any derived MPI type can be used as an etype. Since by now you ought to know how complex derived MPI types can be, you should appreciate how rich a structure of MPI files can be too.

Etypes can then be organized additionally into a *filetype*. A filetype describes data distribution, in terms of etypes and etype-size holes, within the file. The description given by a filetype can be very complex, much like etypes themselves can get as complex as the context demands, and as the programmer can cope with.

MPI processes which open a shared MPI file acquire their own *views* of that file. A view is what a given process can see inside the file. All I/O operations performed by that process occur within the view. Normally one would design the whole I/O in such a way that the views of separate processes would not overlap, but they can overlap.

In this section we will first look at how to manipulate MPI files. Then we'll discuss reading data from and writing data to MPI files. Then we're going to discuss *filetype constructors*, which are in a category similar to MPI datatype constructors, and, finally, we'll have a look at some examples.

On our SP system you can call MPI/IO functions against two file systems. The first one is GPFS. If you use GPFS then all processes that your job runs on *must* be GPFS clients and *must* mount the same GPFS file system, and, of course, this is the file system that you will write to. The other file system you can call MPI/IO functions against is HPSS. You can perform MPI/IO transactions with HPSS from any SP node that has DCE and Encina libraries installed and configured.

At present GPFS is available on all nodes that run parallel production jobs, i.e., nodes that support classes pa and pb. But *not* on the test nodes, which, of course, makes testing MPI/IO jobs somewhat difficult.

GPFS MPI/IO programs don't require linking with any special libraries other than what you normally get if you call the mpcc or the mpxlf90 wrappers. HPSS MPI/IO programs need to be linked with:

```
libmpioapi.a libhpss.a libmpi.a libEncina.a libEncClient.a
libdce.a libdcepthreads.a libpthreads_compat.a libpthreads.a
```

in this order. You will also have to

1. generate your keytab file with rgy_edit
2. define MPIO_LOGIN_NAME in your environment, it should be set to your HPSS user name
3. define MPIO_KEYTAB_PATH and point it to where you store the keytab file
4. define HPSS_LS_SERVER and point it to the HPSS Location Server
5. set MPIO_DEBUG to whatever level of MPI/IO debugging messages you want to receive.
6. use #include <mpio.h> in your program and point to the HPSS version of MPI/IO includes

9.7.1 Manipulating MPI Files

MPI files are not like ordinary files and so MPI provides special functions for opening them, closing them and doing other things with them.

The first thing you always want to do with a file is to open it. The call to do that in C is:

```
int MPI_File_open (MPI_Comm comm, char *filename, int amode, MPI_Info info,
                   MPI_File *fh);
```

and in Fortran

```
MPI_FILE_OPEN(INTEGER COMM, CHARACTER FILENAME(*), INTEGER AMODE, &
              INTEGER INFO, INTEGER FH, INTEGER IERROR)
```

This call must be issued by *all* processes participating in the communicator. It is a blocking call and a barrier call. This call sets a default *view* of the file, about which more later.

There are 3 arguments in this call the likes of which we haven't encountered yet. The first one is amode, which stands for the access mode. All processes opening the file *must* open it in the same access mode. The access modes can be as follows

MPI_MODE_RDONLY— read only

MPI_MODE_RDWR— read and write

MPI_MODE_WRONLY— write only

MPI_MODE_CREATE— create the file if it does not exist

MPI_MODE_EXCL— raise an error if the file already exists and `MPI_MODE_CREATE` is specified

MPI_MODE_DELETE_ON_CLOSE— delete file on close

MPI_MODE_UNIQUE_OPEN— file will not be opened concurrently elsewhere

MPI_MODE_SEQUENTIAL— file will only be accessed sequentially

MPI_MODE_APPEND— set initial position of all file pointers to end of file

They can be combined like this in C:

```
MPI_MODE_WRONLY | MPI_MODE_CREATE | MPI_MODE_EXCL
```

and in Fortran you can use:

```
IOR(MPI_MODE_WRONLY, MPI_MODE_CREATE)
```

The second argument that is specific to `MPI_FILE_OPEN` is `info`. If you use GPFS to write on then this parameter should be left empty. Simply specify:

```
info = MPI_INFO_NULL
```

If you use HPSS then there is quite a lot that you can put in `info`. The `info` object can be loaded with information about how the file should be striped, what permissions it should be created with, how many parallel processes will typically access it, and so on. You can also load it with HPSS specific information of which the most important is HPSS class of service, HPSS type of storage class, and hints about the anticipated size of the file.

The info object is created by calling function

```
int MPI_Info_create(MPI_Info *info);
```

in C and... there is no support for Fortran interface in the HPSS version of MPI/IO. Once you have created your info object then you can begin loading it by calling function `MPI_Info_set`. For example:

```
MPI_Info_set(info, "hpss_cos", "3");
MPI_Info_set(info, "hpss_sclasstype", "DISK");
MPI_Info_set(info, "access_style", "write_once");
```

Once you no longer need an info object you can free it with

```
MPI_Info_free(info);
```

There are many other things you can do with an info object. You can enquire about the number of *key-value* pairs. You can enquire about the keys and then ask about the values. You can duplicate an info object and you can modify any value in a selected *key-value* pair. There are functions for all that. But, since none of that stuff is currently supported on GPFS, we won't dwell on it any more.

`MPI_File_open` is a blocking call. The function does not return until the file has been opened and the file handle returned. This may take a very long time, especially on a parallel file system, and an even longer time with HPSS, where numerous transactions have to be exchanged with Encina in order to open a new file. It would be good if you could send a *file open* request to the system and then go away and do other things, occasionally checking if the file is ready.

Once you have opened a file you will probably write on it or read from it, but this is a complex affair in MPI, so we'll postpone the discussion of this until later.

To close an MPI file simply say:

```
int MPI_File_Close (MPI_File *fh);
```

in C and in Fortran

```
MPI_FILE_CLOSE (FH, IERROR)
INTEGER FH, IERROR
```

You *must* ensure that *all* requests associated with `fh` have completed before you call `MPI_File_Close`. This is a collective barrier operation.

To delete an MPI file call function

```
int MPI_File_delete (char *filename, MPI_Info info);
```

in C and

```
MPI_FILE_DELETE(FILENAME, INFO, IERROR)
CHARACTER(*) FILENAME
INTEGER INFO, IERROR
```

As with `open` you can pass various hints to the file system you work with by loading them into `info`. GPFS, as I said before, doesn't support this feature at this stage, so here you have to use `MPI_INFO_NULL`. And HPSS doesn't have any special hints for deleting either, so here again, you just use `MPI_INFO_NULL`.

MPI files can be statically sized. This is important because if multiple processes write on the same file in parallel, you must have a static file map, so that the processes don't write on each other's blocks. You set a size of an MPI file with

```
int MPI_File_set_size (MPI_File fh, MPI_Offset size);
```

in C and

```
MPI_FILE_SET_SIZE (FH, SIZE, IERROR)
INTEGER FH
INTEGER(KIND=MPI_OFFSET_KIND) SIZE,
INTEGER IERROR
```

in Fortran. Observe that `size` is of type `MPI_Offset`. This is going to be a 64 bit integer on GPFS and HPSS (`long long` in C and `INTEGER(KIND=8)` in Fortran). There must be *no* pending I/O operations when you call this function. You can use this function both to increase and to decrease the size of the file.

If you have a file which is already opened and sized and don't remember what the size was use

```
int MPI_File_get_size (MPI_File fh, MPI_Offset size);
```

in C and

```
MPI_FILE_GET_SIZE (FH, SIZE, IERROR)
INTEGER FH
INTEGER(KIND=MPI_OFFSET_KIND) SIZE,
INTEGER IERROR
```

in Fortran.

You can also alter an `info` record on a file by calling `MPI_File_set_info` and you can read the info record associated with an open file by calling `MPI_File_get_info`.

9.7.2 Writing and Reading MPI Files

Before you can begin writing on an MPI file in parallel, each process participating in the operation must acquire its own *view* of that file. A view is defined in terms of 3 parameters: a displacement, which is a location in the file given as the number of bytes from the beginning of the file, an elementary data type, the *etype*, and a *filetype* about which more below.

This business about views, filetypes and etypes is a little hard to understand without an example. Assume that we have some etype such as, e.g., a particle structure. This is a record that comprises a number of doubles, some integers, and some characters. We have seen how to build the corresponding MPI derived data type in one of the previous sections. Now, let us build a new MPI derived data type, which, say, picks up a second and third particle from an array of 6 particles. Symbolically we can write it as follows:

```
X
OXXOOO
```

where `X` in the first row stands for the etype and the second row represents the new derived data type with one hole, `O`, in front, then two particles, `XX`, and then 3 holes, `OOO`.

Let us define three derived MPI types as follows:

```
type1 = XXOOOO
type2 = OOXXOO
type3 = OOOOXX
```

A *view*, as I have said above, is a triple `(displacement, etype, filetype)`. Define the following three views:

```
(0, X, XXOOOO)
(0, X, OOXXOO)
(0, X, OOOOXX)
```

If these are the views that correspond to three different processes, when a parallel read takes place, the first process will read the first two particles, the second process will read particles 3 and 4 and the third process will read particles 5 and 6. Then the pointer is advanced to the beginning of the next *filetype* item and the read operation can commence. The view of the file that the first process has is:

```
XXOOOO XXOOOO XXOOOO XXOOOO ...
```

The second process sees the following data on the file:

```
OOXXOO OOXXOO OOXXOO OOXXOO ...
```

And the third process' view is:

```
OOOOXX OOOOXX OOOOXX OOOOXX ...
```

In this example all processes' view has the same displacement, but the filetypes are different. A similar effect same can be accomplished by giving 3 different displacements and sharing the same file view:

```
(0, X, XXOOOO)
(sizeof(XX), X, XXOOOO)
(sizeof(XXXX), X, XXOOOO)
```

In summary: in order to avoid stepping on each other's toes, each process must have a different view of the shared file. If the views are constructed soundly, then each process is going to work on a different portion of data.

So how do you construct a view? Use function:

```
int MPI_File_set_view (MPI_File fh, MPI_Offset displacement,
                       MPI_Datatype etype, MPI_Datatype filetype,
                       char *datarep, MPI_Info info);
```

in C and

```
MPI_FILE_SET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, INFO, IERROR)
INTEGER FH, ETYPE, FILETYPE, INFO, IERROR
CHARACTER*(*) DATAREP
INTEGER(KIND=MPI_OFFSET_KIND) DISP
```

in Fortran.

There is one parameter here in these interfaces, which I haven't talked about yet. It is the Data Representation parameter, which is a string.

MPI guarantees full interoperability within a single MPI environment, but there is little support in it, as yet, for external data representation. Yet, the moment you begin writing MPI files, this issue gains in importance, because you are quite likely to process those files on a variety of architectures. The following predefined Data Representation strings are currently available

"native" Data is stored on a file the same way it is stored in memory. This is very fast but non-portable. Use it when writing scratch files.

"internal" This is a portable data format, which is supported across various platforms by a given MPI implementation, e.g., MPICH. You may not be able, in principle, to write data in this format with MPICH and then read it with LAM MPI or with IBM MPI. But you should be able to write data in this format with MPICH on Solaris and then read it, say, on DEC Alpha.

"external32" All data is converted to and from "external32". Should work from MPI to MPI and from vendor to vendor. But data precision is lost (to 32-bits only), and I/O should be expected to suffer.

Once you've set a view on a file, you can also get it back with function

```
int MPI_File_get_view (MPI_File fh, MPI_Offset *displacement,
                       MPI_Datatype *etype, MPI_Datatype *filetype,
                       char *datarep);
```

in C and similarly in Fortran.

So, at this stage all processes should have opened a file and should have defined their view on that file. Now we can begin to write data to the file and to read data from it.

Assuming that you have structure the data on the file with etype and filetype definitions the simplest way to write data on a file is to call function

```
int MPI_File_write (MPI_File fh, void *buffer, int count,
                    MPI_Datatype datatype, MPI_Status *status);
```

in C and

```
MPI_FILE_WRITE(FH, BUFFER, COUNT, DATATYPE, STATUS, IERROR)
<type> BUF(*)
INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
```

This function transfers `count` data items of type datatype from a buffer pointed to by `buffer` to file `fh`. The data will be written at a position in the file pointed to by the file pointer. This operation will advance the pointer according to the formula:

$$new\_file\_offset = old\_file\_offset + \frac{elements(datatype)}{elements(etype)} \times count$$

If `datatype` is the same as `filetype`, which is a sensible thing to do, then the pointer will get advanced, in units of etype, by `count filetypes`, so that, in effect, the reading of the file will proceed as in the example discussed above.

Once you've written some data on the file you can read it back with

```
int MPI_File_read(MPI_File fh, void *buf, int count,
                  MPI_Datatype datatype, MPI_Status *status)
```

in C and with

```
MPI_FILE_READ(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
<type> BUF(*)
INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
```

in Fortran.

These two functions, `MPI_File_write` and `MPI_File_read` are blocking and non-collective, i.e., each process does the reads on its own. Each process can read the file differently and in its own way and time. There is no barrier. Some processes may choose to read their data chunks from the file, some may forgo reading altogether, depending on what they do.

There is a collective version of these calls, which forces *all* processes in the communicator to read data simultaneously and to wait for each other. These collective functions are called `MPI_File_read_all` and `MPI_File_write_all` and their synopsis (though not their semantics) is the same as for the non-collective versions.

There are also non-blocking versions of of these functions. They are:

```
int MPI_File_iwrite(MPI_File fh, void *buf, int count,
                    MPI_Datatype datatype, MPI_Request *request)
MPI_FILE_IWRITE(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
<type> BUF(*)
INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
```

and

```
int MPI_File_iread(MPI_File fh, void *buf, int count,
                   MPI_Datatype datatype, MPI_Request *request)
MPI_FILE_IREAD(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
<type> BUF(*)
INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
```

As you see the list of parameters is the same with the exception that `status` is replaced with `request`. You have to keep inspecting the `request` to check if the operation has completed. Then you can inspect the status with another MPI function.

These non-blocking writes and reads are very useful. Any external I/O operations are excruciatingly slow compared with memory access or with operations that are done on the registers. Consequently if you can organise your program so that you issue a non-blocking I/O request in advance, then go back to your computations and keep checking every now and then if the I/O operation completed, you'll be able to mask the slowness of I/O with computations. Programs like that can be very fast. But they are also extremely difficult to write and to debug.

The functions discussed so far perform sequential writes within their respective views. What if you want to write data at various locations within your view jumping here and there out of order?

For this you would use a family of functions with the extension _AT. These functions are like the functions already discussed, but they take one more parameter, namely the offset from the beginning of the view.

File offsets in MPI/IO are always given in terms of etypes and are always measured from the beginning of the *view*. This is a matter of semantics, naming, and to agree on this simply saves unnecessary confusion. File displacements on the other hand are given in bytes and are measured from the beginning of the file.

The synopsis for the _AT functions is as follows:

```
int MPI_File_write_at(MPI_File fh, MPI_Offset offset, void *buf,
                      int count, MPI_Datatype datatype, MPI_Status *status)

MPI_FILE_WRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
<type> BUF(*)
INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET

int MPI_File_read_at(MPI_File fh, MPI_Offset offset, void *buf,
                     int count, MPI_Datatype datatype, MPI_Status *status)

MPI_FILE_READ_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
<type> BUF(*)
INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
```

and similarly for the nonblocking and collective versions.

There is one more group of MPI reads and writes. For all functions discussed above every process would maintain its own file pointer. In the _AT functions that pointer would be manipulated explicitly. in MPI_FILE_READ it would be advanced implicitly. But every process would end up reading different data.

What if we want all processes to read the same data from the same file?

In this case we need to use data access functions with shared file pointers. The functions are:

```
int MPI_File_write_shared(MPI_File fh, void *buf, int count,
                          MPI_Datatype datatype, MPI_Status *status)
```

```
MPI_FILE_WRITE_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
<type> BUF(*)
INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR

int MPI_File_read_shared(MPI_File fh, void *buf, int count,
                         MPI_Datatype datatype, MPI_Status *status)
MPI_FILE_READ_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
<type> BUF(*)
INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
```

and they also have their collective and non-blocking counterparts.

9.7.3 File Consistency

Files that are manipulated simultaneously by multiple processes *without locking* can easily become corrupted. It is therefore vital that MPI I/O semantics are defined so that this corruption can be avoided. It is possible to construct overlapping file views and this at times may not be accidental. If this is the case then the programmer may need to order writes on a file by setting barriers or synchronizing operations with messages. It is also possible to set the so called *atomic* mode on a file, and this will result in all I/O operations on that file becoming *sequentially consistent* although the sequence itself will remain undetermined. The atomic mode is set with a call to

```
int MPI_File_set_atomicity(MPI_File fh, int flag)

MPI_FILE_SET_ATOMICITY(FH, FLAG, IERROR)
INTEGER FH, IERROR
LOGICAL FLAG
```

Additionally MPI files can be *synced* or *flushed* with a call to:

```
int MPI_File_sync(MPI_File fh)

MPI_FILE_SYNC(FH, IERROR)
INTEGER FH, IERROR
```

The HPSS version of MPI allows to read files directly from tapes. Such files are referred to as *sequential stream files*. They must be opened with the `MPI_MODE_SEQUENTIAL` flag set in the amode. For these files, the only permitted data access operations are shared file pointer reads and writes. Filetypes and etypes with holes are erroneous.

9.7.4 Logical versus Physical File Layout

MPI specifies how the data should be laid out in a virtual file structure (the view), not how that file structure is to be stored on one or more disks. Specification of the physical file structure was avoided because mapping of files to disks is system specific, and any specific control over file layout would therefore restrict program portability. However, there are still cases where some information may be necessary to optimize file layout. This information can be provided as hints specified via info when a file is created.

9.8 Dynamic Process Creation

9.9 Environmental Inquiries

9.10 Error Handling

9.11 Profiling

Index